Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: 3 August 2026

T. Geoghegan
ISRG
C. Patton
Cloudflare
B. Pitman
ISRG
E. Rescorla
Independent
C. A. Wood
Cloudflare

30 January 2026

Distributed Aggregation Protocol for Privacy Preserving Measurement
draft-ietf-ppm-dap-17

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 on some server, 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 which can be used to collect aggregate data without revealing any individual contributor'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 3 August 2026.

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 aggregation servers. The aggregators' goal is to compute some aggregate statistic over measurements generated by clients without learning the measurements 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 measurement is ever seen in the

clear by any aggregator.

1.1. Change Log

(RFC EDITOR: Remove this section.)

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

17:

* Bump version tag from "dap-16" to "dap-17". (*)

* Align IANA considerations with RFC 8126 and https://www.iana.org/help/protocol-registration.

* Add RFC Editor notes to change domain separation tags from "dap-17" to "dap" on RFC publication.

* Fix definitions of time types. (#759)

* Rename VDAF preparation to verification. (#752)

* Simplify media types. (*) (#748)

* Bump draft-irtf-cfrg-vdaf to -18 ([VDAF]).

16:

* Bump draft-irtf-cfrg-vdaf-13 to 15 [VDAF] and adopt changes to the ping-pong API. (#705, #718)

* Allow many reports to be uploaded at once. (*) (#686)

* Remove TLS presentation language syntax extensions. (#707)

* Use HTTP message content length to determine length of vectors in AggregationJobInitReq, AggregationJobResp and AggregationJobContinueReq messages. (*) (#717)

* Represent the Time and Duration types as a number of time_precision intervals, rather than seconds (*) (#720).

* Discuss the property of "verifiability" instead of "robustness" to match recent VDAF changes (https://github.com/cfrg/draft-irtf-cfrg-vdaf/pull/558). (#725)

* Bump version tag from "dap-15" to "dap-16". (*)

15:

* Specify body of responses to aggregation job GET requests. (#651)

* Add diagram illustrating object lifecycles and relationships. (#655)

* Use aasvg for prettier diagrams. (#657)

* Add more precise description of time and intervals. (#658)

* Reorganize text for clarity and flow. (#659, #660, #661, #663, #665, #666, #668, #672, #678, #680, #684, #653, #654)

* Align with RFC 9205 recommendations. (*) (#673, #683)

* Define consistent semantics for long-running interactions: aggregation jobs, collection jobs and aggregate shares. (*) (#674, #675, #677)

* Add security consideration for predictable task IDs. (#679)

* Bump version tag from "dap-14" to "dap-15". (*)

14:

* Enforce VDAF aggregation parameter validity. This is not relevant for Prio3, which requires only that reports be aggregated at most once. It is relevant for VDAFs for which validity depends on how many times a report might be aggregated (at most once in DAP). (*)

* Require all timestamps to be truncated by time_precision. (*)

* Bump draft-irtf-cfrg-vdaf-13 to 14 [VDAF]. There are no functional or breaking changes in this draft.

* Clarify conditions for rejecting reports based on the report metadata, including the timestamp and public and private extensions.

* Clarify that the Helper responds with 202 Accepted to an aggregation continuation request.

* Bump version tag from "dap-13" to "dap-14". (*)

13:

* Bump draft-irtf-cfrg-vdaf-12 to 13 [VDAF] and adopt the streaming aggregation interface. Accordingly, clarify that DAP is only compatible with VDAFs for which aggregation is order insensitive.

* Add public extensions to report metadata. (*)

* Improve extension points for batch modes. (*)

* During the upload interaction, allow the Leader to indicate to the Client which set of report extensions it doesn’t support.

* Add a start time to task parameters and require rejection of reports outside of the time validity window. Incidentally, replace the task end time with a task duration parameter.

* Clarify underspecified behavior around aggregation skew recovery.

* Improve IANA considerations and add guidelines for extending DAP.

* Rename "upload extension" to "report extension", and "prepare error" to "report error", to better align the names of these types with their functionality.

* Bump version tag from "dap-12" to "dap-13". (*)

12:

* Bump draft-irtf-cfrg-vdaf-08 to 12 [VDAF], and specify the newly-defined application context string to be a concatenation of the DAP version in use with the task ID. (*)

* Add support for "asynchronous" aggregation, based on the Leader polling the Helper for the result of each step of aggregation. (*)

* Update collection semantics to match the new aggregation semantics introduced in support of asynchronous aggregation. (*)

* Clarify the requirements around report replay protection, defining when and how report IDs must be checked and stored in order to correctly detect replays.

* Remove support for per-task HPKE configurations. (*)

* Rename "query type" to "batch mode", to align the name of this configuration value with its functionality.

* Rename the "fixed-size" batch mode to "leader-selected", to align the name with the behavior of this query type.

* Remove the max_batch_size parameter of the "fixed-size" batch mode.

* Restore the part_batch_selector field of the Collection structure, which was removed in draft 11, as it is required to decrypt collection results in some cases. (*)

* Update PrepareError allocations in order to remove an unused value and to reserve the zero value for testing. (*)

* Document distributed-system and synchronization concerns in the operational considerations section.

* Document additional storage and runtime requirements in the operational considerations section.

* Document deviations from the presentation language of Section 3 of [RFC8446] for structures described in this specification.

* Clarify that differential privacy mitigations can help with privacy, rather than robustness, in the operational considerations section.

* Bump version tag from "dap-11" to "dap-12". (*)

11:

* Remove support for multi-collection of batches, as well as the fixed-size query type’s by_batch_id query. (*)

* Clarify purpose of report ID uniqueness.

* Bump version tag from "dap-10" to "dap-11". (*)

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.

All examples in this document wrap long lines using the Single Backslash Strategy of [RFC8792], Section 7.

1.2.1. Glossary of Terms

Aggregate result: The output of the aggregation function. As defined in [VDAF].

Aggregate share: A secret share of the aggregate result computed by each Aggregator and transmitted to the Collector. As defined in [VDAF].

Aggregation function: The function computed over the measurements generated by the Clients and the aggregation parameter selected by the Collector. As defined in [VDAF].

Aggregation parameter: Parameter selected by the Collector used to refine a batch of measurements for aggregation. As defined in [VDAF].

Aggregator: The party that receives report shares from Clients and validates and aggregates them with the help of the other Aggregator, producing aggregate shares for the Collector. As defined in [VDAF].

Batch: A set of reports (i.e., measurements) that are aggregated into an aggregate result. As defined in [VDAF].

Batch bucket: State associated with a given batch allowing the aggregators to perform incremental aggregation. Depending on the batch mode, there may be many batch buckets tracking the state of a single 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 party that generates a measurement and uploads a report, as defined in [VDAF]. 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 assembles the aggregate result from the aggregate shares constructed by the Aggregators. As defined in [VDAF].

Helper: The Aggregator that executes the aggregation and collection interactions initiated 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 VDAF verification. Many output shares are combined into an aggregate share during VDAF aggregation. 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 that must be aggregated before a batch can be collected.

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

Report: A cryptographically protected measurement uploaded to the Leader by a Client. Includes a public share and a pair of report shares, one for each Aggregator.

Report share: An input share encrypted under the HPKE public key of an Aggregator [HPKE]. The report share also includes some associated data used for processing the report.

Task: A set of measurements of an understood type which will be reported by the Clients, aggregated by the Aggregators and received by the Collector. Many collections can be performed in the course of a single task.

2. Overview

The protocol is executed by a large set of Clients and a pair of Aggregators. Each Client's input to the protocol is its measurement (or set of measurements, e.g., counts of some user behavior). Given a set of measurements meas_1, ..., meas_N held by the Clients, and an "aggregation parameter" agg_param shared by the Aggregators, the goal of DAP is to compute agg_result = F(agg_param, meas_1, ..., meas_N) for some function F while revealing nothing else about the measurements. We call F the "aggregation function" and agg_result the "aggregate result".

(RFC EDITOR: Please update the normative reference to VDAF in the next paragraph to the VDAF RFC during AUTH48. We hope that VDAF will have been published by then.)

DAP is extensible in that it allows for the addition of new cryptographic schemes that compute different aggregation functions, determined by the Verifiable Distributed Aggregation Function, or [VDAF], used to compute it.

VDAFs rely on secret sharing to protect the privacy of the measurements. Rather than sending its measurement 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 scheme has two important properties:

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

* Aggregators can compute secret shares of the aggregate result by aggregating their shares locally into "aggregate shares", which may later be merged into the aggregate result.

DAP is not compatible with all VDAFs. DAP only supports VDAFs whose aggregation results are independent of the order in which measurements are aggregated (see Section 4.4.1 of [VDAF]). Some VDAFs may involve three or more Aggregators, but DAP requires exactly two Aggregators. Some VDAFs allow measurements to be aggregated multiple times with a different aggregation parameter. DAP may be compatible with such VDAFs, but only allows each measurement to be aggregated once.

2.1. System Architecture

The basic unit of DAP operation is the _task_ (Section 4.2), which corresponds to a set of measurements of a single type. A given task may result in multiple aggregated reported results, for instance when measurements are collected over a long time period and broken up into multiple batches according to different time windows.

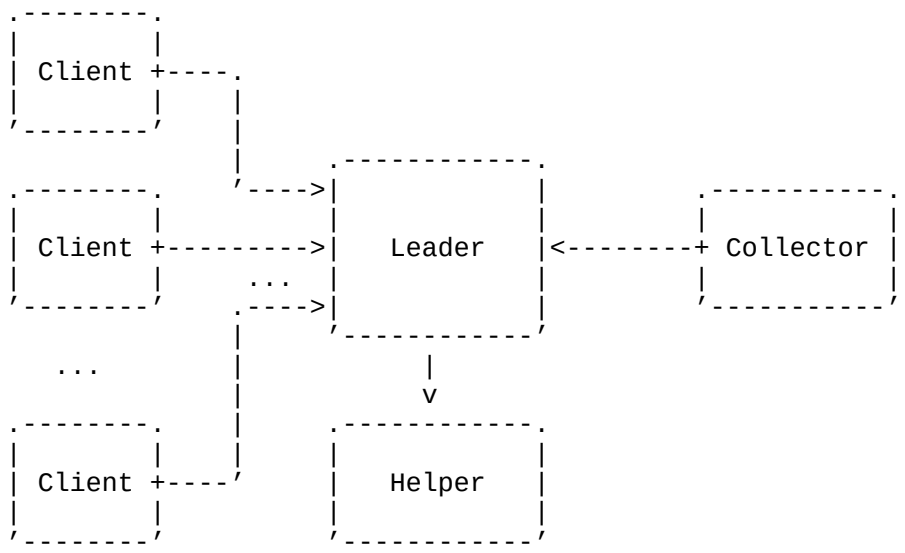

Figure 1: DAP architecture

The main participants in the protocol are as follows:

Collector: The party which wants to obtain the aggregate result over the measurements generated by the Clients. A task will have a single Collector.

Clients: The parties which take the measurements and report them to the Aggregators. In order to provide reasonable levels of privacy, there must be a large number of Clients.

Leader: The Aggregator responsible for coordinating the protocol. It receives the reports from Clients, aggregates them with the assistance of the Helper, and it orchestrates the process of computing the aggregate result as requested by the Collector. Each task has a single Leader.

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. Each task has a single Helper.

Figure 1 illustrates which participants exchange HTTP messages. Arrows go from HTTP clients to HTTP servers. Some DAP participants may be HTTP clients sometimes but HTTP servers at other times. It is even possible for a single entity to perform multiple DAP roles. For example, the Collector could also be one of the Aggregators.

In the course of a measurement task, each Client records its own measurement, packages it up into a report, and sends it to the Leader. Each share is encrypted to only one of the two Aggregators so that even though both pass through the Leader, the Leader is unable to see or modify the Helper's share. 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 and assembling them into a aggregate shares for the Collector. Depending on the VDAF, it may be possible to process each report as it is uploaded, or it may be necessary to wait until the Collector initializes a collection job before processing can begin.

2.2. Validating Measurements

An essential goal of any data collection pipeline is ensuring that the data being aggregated is "valid". For example, each measurement might be expected to be a number between 0 and 10. 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. To an Aggregator, a secret share of a valid measurement is indistinguishable from a secret share of an invalid measurement.

DAP validates inputs using an interactive computation between the Leader and Helper. At the beginning of this computation, each Aggregator holds an input share uploaded by the Client. At the end of the computation, each Aggregator either obtains an output share that is ready to be aggregated or rejects the report as invalid.

This process is known as "verification" and is specified by the VDAF itself (Section 5.2 of [VDAF]). The report generated by the Client includes information used by the Aggregators to verify the report. For example, Prio3 (Section 7 of [VDAF]) includes a zero-knowledge proof of the measurement's validity (Section 7.1 of [VDAF]). Verifying this proof reveals nothing about the underlying measurement but its validity.

The specific properties attested to by the proof depend on the measurement being taken. For instance, if the task is measuring the latency of some operation, the proof might demonstrate that the value reported was between 0 and 60 seconds. But to report which of N options a user selected, the report might contain N integers and the proof would demonstrate that N-1 of them were 0 and the other was 1.

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

2.3. Replay Protection and Double Collection

Another goal of DAP is to mitigate replay attacks in which a report is aggregated in multiple batches or multiple times in a single batch. This would allow the attacker to learn more information about the underlying measurement than it would otherwise.

When a Client generates a report, it also generates a random nonce, called the "report ID". Each Aggregator is responsible for storing the IDs of reports it has aggregated and rejecting replayed reports.

DAP must also ensure that any batch is only collected once, even if new reports arrive that would fall into that batch. Otherwise, comparing the new aggregate result to the previous aggregate result can violate the privacy of the added reports.

Aggregators are responsible for refusing new reports if the batch they fall into has been collected (Section 4.6).

2.4. Lifecycle of Protocol Objects

The following diagrams illustrate how the various objects in the protocol are constructed or transformed into other protocol objects. Oval nodes are verbs or actions which process, transform or combine one or more objects into one or more other objects.

The diagrams in this section do not necessarily illustrate how participants communicate. In particular, the processing of aggregation jobs happens in distinct, non-colluding parties.

2.4.1. The Upload Interaction

Reports are 1 to 1 with measurements. In this illustration, i distinct Clients upload i distinct reports, but a single Client could upload multiple reports to a task (see Section 8.1 for some implications of this). The process of sharding measurements, constructing reports and uploading them is specified in Section 4.4.

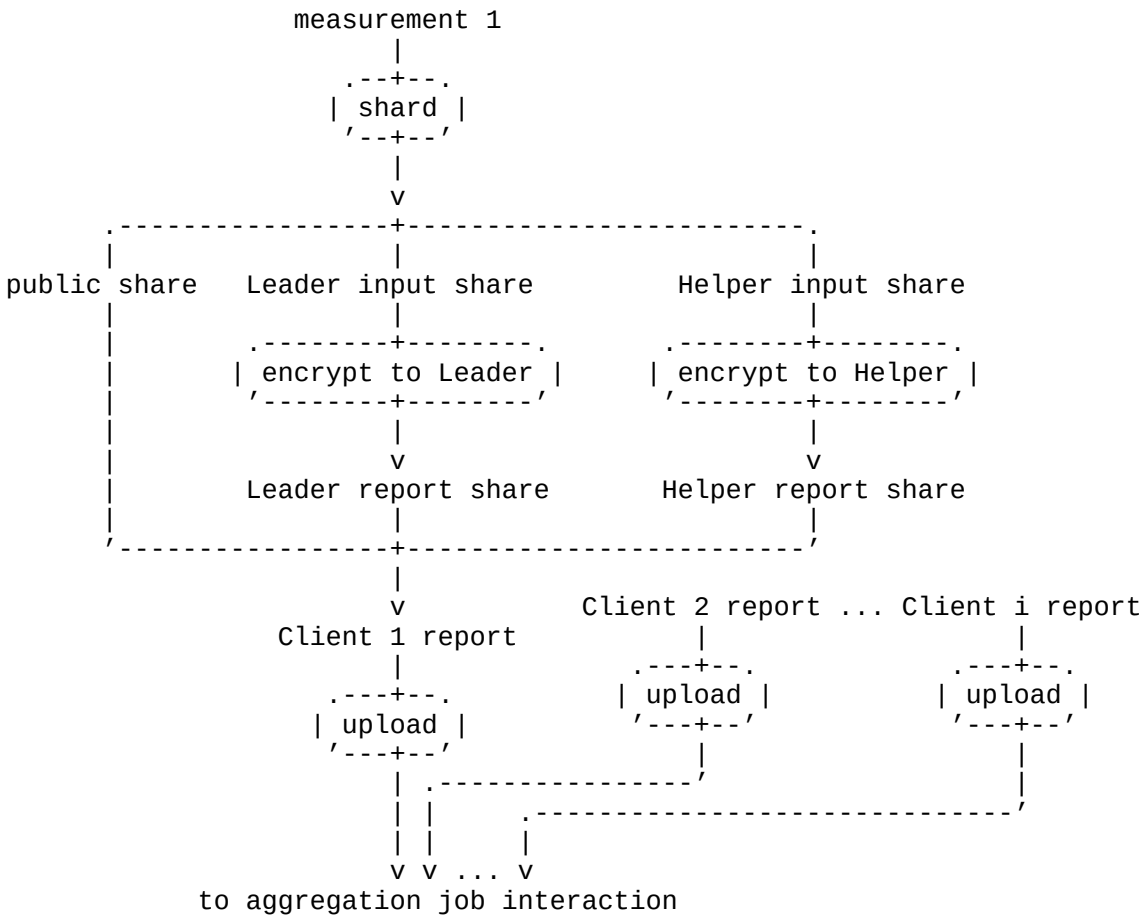

Figure 2: Lifecycles of protocol objects in the upload interaction

2.4.2. The Aggregation Interaction

Reports are many to 1 with aggregation jobs. The Leader assigns each of the i reports into one of j different aggregation jobs, which can be run in parallel by the Aggregators. Each aggregation job verifies k reports, outputting k output shares. k is roughly i / j, but it is not necessary for aggregation jobs to have uniform size.

See Section 4.5 for the specification of aggregation jobs and Section 6 for some discussion of aggregation job scheduling strategies and their performance implications.

Output shares are accumulated into m batch buckets, and so have a many to 1 relationship. Aggregation jobs and batch buckets are not necessarily 1 to 1. A single aggregation job may contribute to multiple batch buckets, and multiple aggregation jobs may contribute to the same batch bucket.

The assignation of output shares to batch buckets is specified in Section 4.5.3.3.

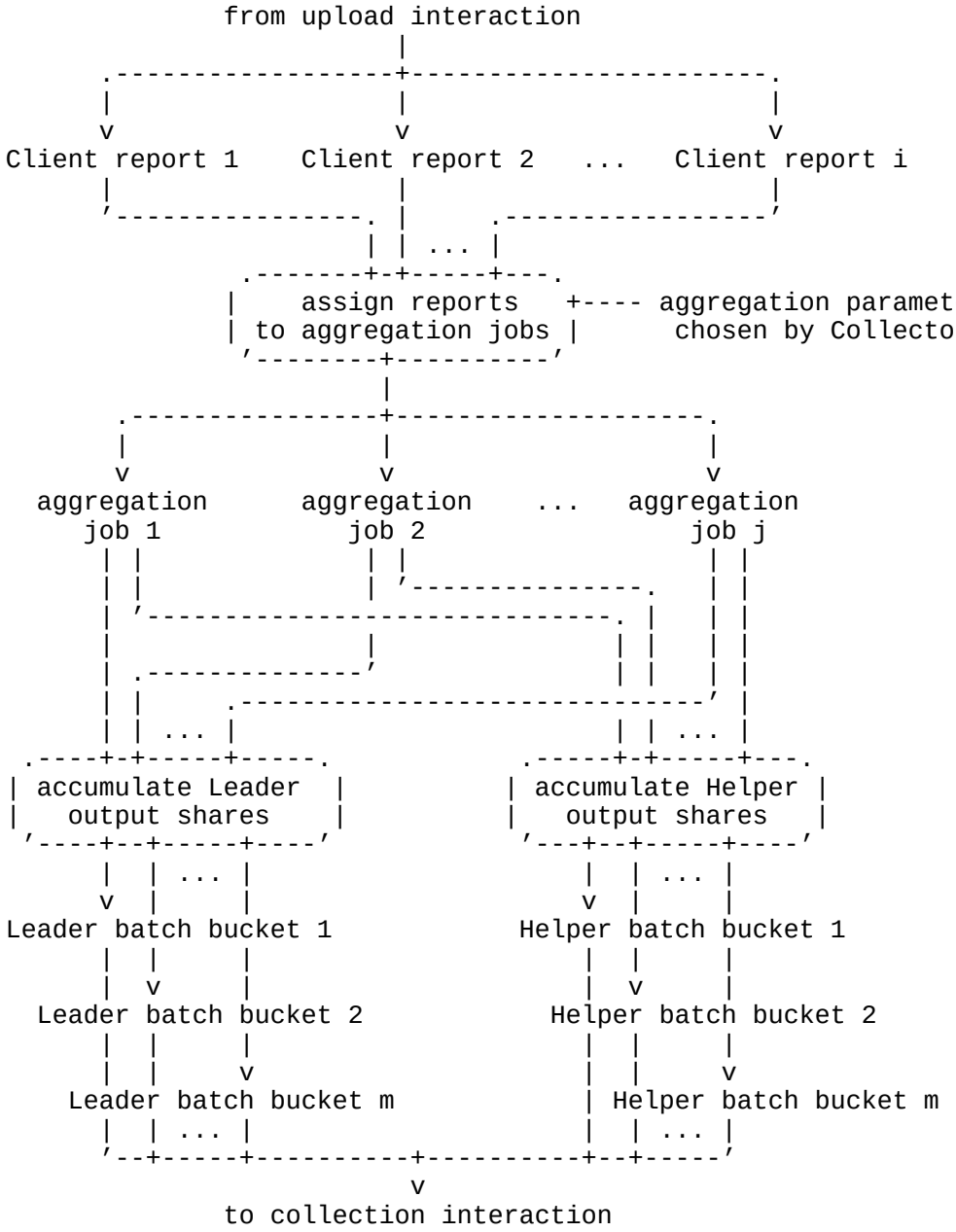

Figure 3: Lifecycles of protocol objects in the aggregation job interaction

Each aggregation job verifies k reports (each consisting of a public share, Leader report share and Helper report share) into k Leader output shares and k Helper output shares. The aggregation parameter is chosen by the Collector (or in some settings it may be known prior to the Collector's involvement, as discussed in Section 4.5.1) and is used to verify all the reports in one or more aggregation jobs.

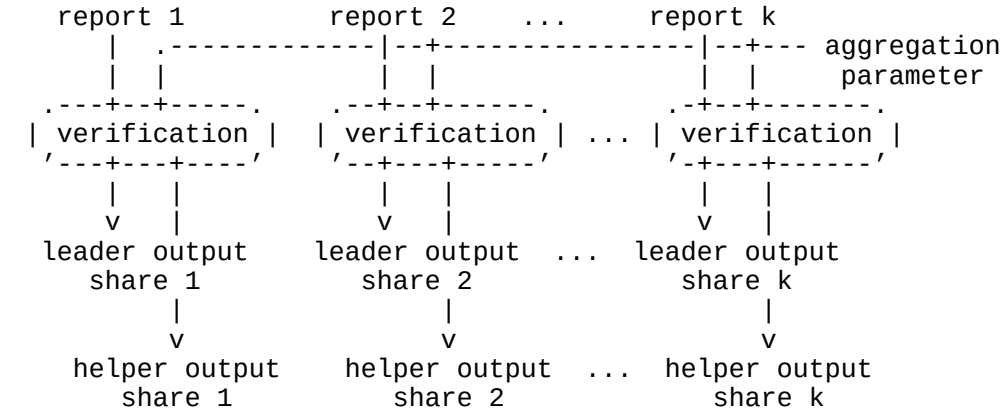

Figure 4: Detail of an individual aggregation job

Report shares, input shares and output shares have a 1 to 1 to 1 relationship. Report shares are decrypted into input shares and then refined into output shares during VDAF verification.

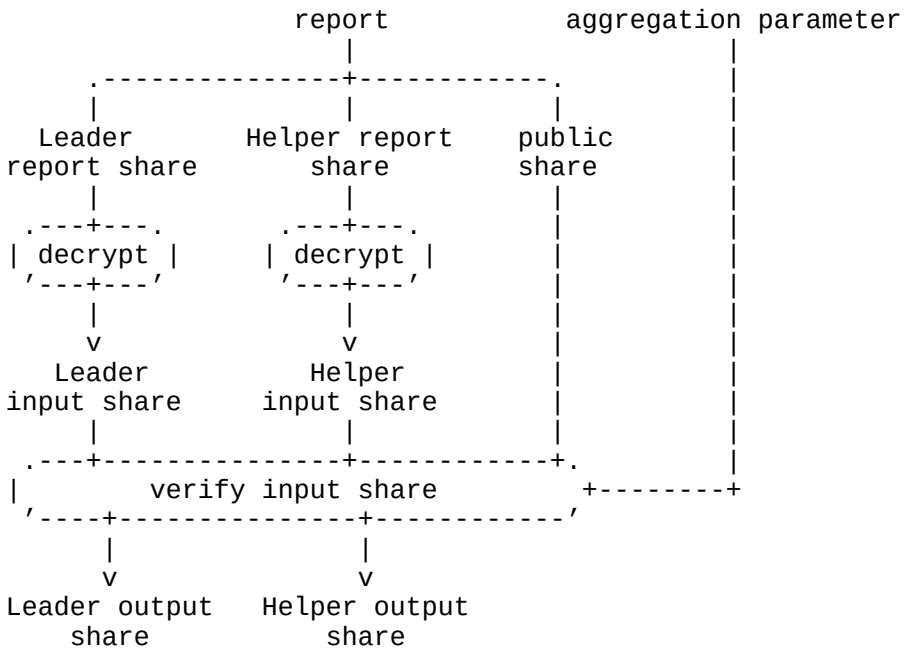

Figure 5: Detail of verification of an individual report

2.4.3. The Collection Interaction

Using the Collector's query, each Aggregator will merge one or more batch buckets together into its aggregate share, meaning batch buckets are many to 1 with aggregate shares.

The Leader and Helper finally deliver their encrypted aggregate shares to the Collector to be decrypted and then unsharded into the aggregate result. Since there are always exactly two Aggregators, aggregate shares are 2 to 1 with aggregate results. The collection interaction is specified in Section 4.6.

There can be many aggregate results for a single task. The Collection process may occur multiple times for each task, with the Collector obtaining multiple aggregate results. For example, imagine tasks where the Collector obtains aggregate results once an hour, or every time 10,000,000 reports are uploaded.

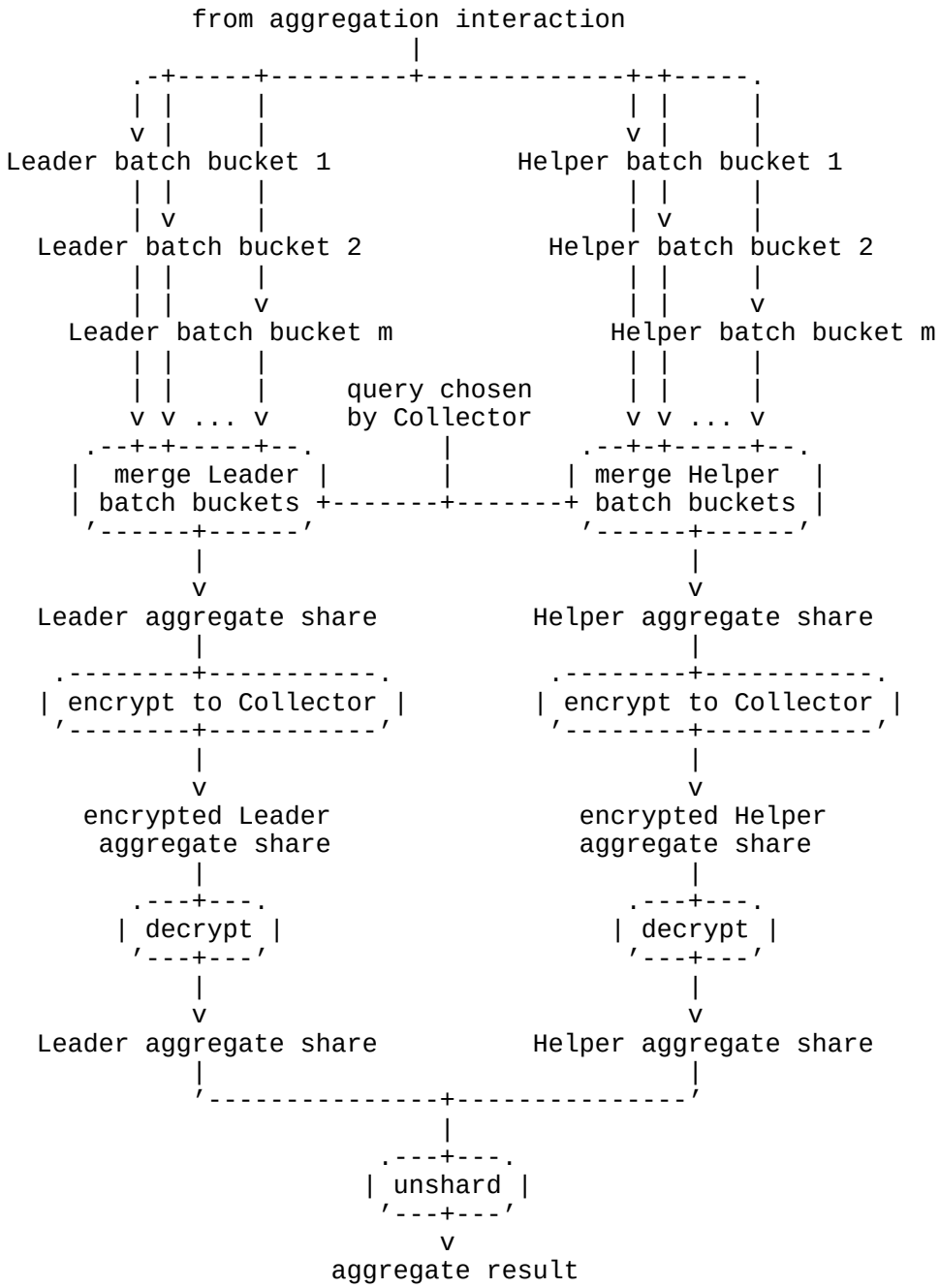

Figure 6: Lifecycles of protocol objects in the collection interaction

3. HTTP Usage

DAP is defined in terms of HTTP [RFC9110] resources. These are HPKE configurations (Section 4.4.1), reports (Section 4.4), aggregation jobs (Section 4.5), collection jobs (Section 4.6), and aggregate shares (Section 4.6.3).

Each resource has a URL. Resource URLs are specified as string literals containing variables. Variables are expanded into strings according to the following rules:

* Variables {leader} and {helper} are replaced with the base API URL of the Leader and Helper respectively.

* Variables {task-id}, {aggregation-job-id}, {aggregate-share-id}, and {collection-job-id} are replaced with the task ID (Section 4.2), aggregation job ID (Section 4.5.2), aggregate share ID (Section 4.6.3) 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 URL {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.

Protocol participants act on resources using HTTP requests, which follow the semantics laid out in [RFC9110], in particular with regard to safety and idempotence of HTTP methods (Sections 9.2.1 and 9.2.2 of [RFC9110], respectively).

The use of HTTPS is REQUIRED to provide server authentication and confidentiality. TLS certificates MUST be checked according to [RFC9110], Section 4.3.4.

3.1. Asynchronous Request Handling

Many of the protocol’s interactions may be handled asynchronously so that servers can appropriately allocate resources for long-running transactions.

In DAP, an HTTP server indicates that it is deferring the handling of a request by immediately sending an empty response body with a successful status code ([RFC9110], Section 15.3). The response SHOULD include a Retry-After field ([RFC9110], Section 10.2.3) to suggest a polling interval to the HTTP client. The HTTP client then polls the state of the resource by sending GET requests to the resource URL. In some interactions, the resource's location will be indicated by a Location header in the HTTP server's response ([RFC9110], Section 10.2.2). Otherwise the resource URL is the URL to which the HTTP client initially sent its request.

The HTTP client SHOULD use each response's Retry-After header field to decide when to fetch the resource. The HTTP server responds the same way as it did to the initial request until either the resource is ready, from which point it responds with the resource's representation ([RFC9110], Section 3.2), or handling the request fails, in which case it MUST abort with the error that caused the failure.

The HTTP server may instead handle the request immediately. It waits to respond to the HTTP client's request until the resource is ready, in which case it responds with the resource's representation, or handling the request fails, in which case it MUST abort with the error that caused the failure.

Implementations are not required to support GET on resources if they are served synchronously, but they could do so, as a way for other protocol participants to retrieve the results of some transaction later on. The retention period for job results is an implementation detail.

3.2. HTTP Status Codes

HTTP servers participating in DAP MAY use any status code from the applicable class when constructing HTTP responses, but HTTP clients MAY treat any status code as the most general code of that class. For example, 202 may be handled as 200, or 499 as 400.

3.3. Presentation Language

We use the presentation language defined in [RFC8446], Section 3 to define messages in the protocol. Encoding and decoding of these messages as byte strings also follows [RFC8446].

3.4. Request Authentication

The protocol is made up of several interactions in which different subsets of participants interact with each other.

In those cases where a channel between two participants is tunneled through another protocol participant, Hybrid Public Key Encryption ([HPKE]) ensures that only the intended recipient can see a message in the clear.

In other cases, HTTP client authentication is required 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 ([RFC9110], Section 11.6.2), which can be added to any protocol message.

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

* [RFC9421] HTTP message signatures authenticate messages without transmitting a secret.

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

Request authentication is REQUIRED in the following interactions:

* Leaders initializing or continuing aggregation jobs with Helpers (Section 4.5).

* Collectors initializing or polling collection jobs with Leaders (Section 4.6.1).

* Leaders obtaining aggregate shares from Helpers (Section 4.6.3).

3.5. Errors

Errors are reported as HTTP status codes. Any of the standard client or server errors (the 4xx or 5xx classes, respectively, from Section 15 of [RFC9110]) are permitted.

When the server responds with an error status code, it SHOULD provide additional information using a problem detail object [RFC9457] in the response body. If the response body does consist of a problem detail object, the status code MUST indicate a client or server error.

To facilitate automatic response to errors, this document defines the following standard tokens for use in the "type" field:

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.
batchInvalid	The batch boundary check for Collector’s query failed.
invalidBatchSize	There are an invalid number of reports in the batch.
invalidAggregationParameter	The aggregation parameter assigned to a batch is invalid.
batchMismatch	Aggregators disagree on the report shares that were aggregated in a batch.
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.
unsupportedExtension	An upload request’s extensions list includes an unknown extension.

Table 1

These types are scoped to the errors sub-namespace of the DAP URN namespace, e.g., urn:ietf:params:ppm:dap:error:invalidMessage.

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 9.3). 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 that the response's status code will indicate a client error unless specified otherwise.

4. Protocol Definition

DAP has three major interactions which need to be defined:

* Clients upload reports to the Aggregators, specified in Section 4.4

* Aggregators jointly verify reports and aggregate them together, specified in Section 4.5

* The Collector collects aggregated results from the Aggregators, specified in Section 4.6

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

4.1. Basic Type Definitions

A ReportID is used to uniquely identify a report in the context of a DAP task.

```
opaque ReportID[16];
```

Role enumerates the roles assumed by protocol participants.

```
enum {
  collector(0),
  client(1),
  leader(2),
  helper(3),
  (255)
} Role;
```

HpkeCiphertext is a message encrypted using [HPKE] and metadata needed to decrypt it. HpkeConfigId identifies a server's HPKE configuration (see Section 4.4.1).

```
uint8 HpkeConfigId;

struct {
  HpkeConfigId config_id;
  opaque enc<1..2^16-1>;
  opaque payload<1..2^32-1>;
} HpkeCiphertext;
```

config_id identifies the HPKE configuration to which the message was encrypted. enc and payload correspond to the values returned by the [HPKE] SealBase() function. Later sections describe how to use SealBase() in different situations.

Empty is a zero-length byte string.

```
struct {} Empty;
```

Errors that occurred while handling individual reports in the upload or aggregation interactions are represented by the following enum:

```
enum {
  reserved(0),
  batch_collected(1),
  report_replayed(2),
  report_dropped(3),
  hpke_unknown_config_id(4),
  hpke_decrypt_error(5),
  vdaf_verify_error(6),
  task_expired(7),
  invalid_message(8),
  report_too_early(9),
  task_not_started(10),
  outdated_config(11),
  (255)
} ReportError;
```

4.1.1. Times, Durations and Intervals

```
uint64 TimePrecision;

uint64 Time;
```

A TimePrecision is an integer number of seconds, used to compute times and durations in DAP. The time precision is a parameter of a task (Section 4.2).

Times are integers, representing a number of TimePrecisions since the Epoch, defined in section 4.16 of [POSIX]. That is, the number of seconds after 1970-01-01 00:00:00 UTC, excluding leap seconds, divided by the task's time_precision.

One POSIX timestamp is said to be before (respectively, after) another POSIX timestamp if it is less than (respectively, greater than) the other value.

Times can only be meaningfully compared to one another if they use the same time precision.

```
uint64 Duration;
```

Durations of time are integers, representing a number of TimePrecisions. That is, a number of seconds divided by the task's time_precision. A duration can be added to a time to produce another time.

```
struct {
  Time start;
  Duration duration;
} Interval;
```

Intervals of time consist of a start time and a duration. Intervals are half-open; that is, start is included and (start + duration) is excluded. A time that is before the start of an Interval is said to be before that interval. A time that is equal to or after Interval.start + Interval.duration is said to be after the interval. A time that is either before or after an interval is said to be outside the interval. A time that is neither before nor after an interval is said to be inside or fall within the interval.

Intervals can only be meaningfully compared to one another if they use the same time precision.

4.1.1.1. Examples

Suppose a task's time_precision is 10 seconds. A Time whose value is 123456789 represents the POSIX timestamp 1234567890, or 2009-02-13 23:31:30 UTC. A Duration whose value is 11 represents a duration of 110 seconds.

An Interval whose start is 123456789 and whose duration is 11 represents the interval from time from POSIX timestamp 1234567890 to 1234568000, or 2009-02-13 23:31:30 UTC to 2009-02-13 23:33:20 UTC.

4.1.2. VDAF Types

The 16-byte ReportID is used as the nonce parameter for the VDAF shard and verify_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
* VerifierShare
* VerifierMessage

4.2. Task Configuration

A task represents a single measurement process, though potentially aggregating multiple, non-overlapping batches of measurements. Each participant in a task must agree on its configuration prior to its execution. This document does not specify a mechanism for distributing task parameters among participants.

A task is uniquely identified by its task ID:

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

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

* The VDAF which determines the type of measurements and the aggregation function. The VDAF itself may have further parameters (e.g., number of buckets in a Prio3Histogram).

* A URL relative to which the Leader's API resources can be found.

* A URL relative to which the Helper's API resources can be found.

* The batch mode for this task (see Section 4.2.1), which determines how reports are grouped into batches.

* task_interval (Interval): Reports whose timestamp is outside of this interval will be rejected by the Aggregators.

* time_precision (TimePrecision): The time precision used in this task. See Section 4.1.1.

The Leader and Helper API URLs MAY include arbitrary path components.

In order to facilitate the aggregation and collection interactions, each of the Aggregators is configured with the following parameters:

* min_batch_size (uint64): The smallest number of reports the batch is allowed to include. A larger minimum batch size will yield a higher degree of privacy. However, this ultimately depends on the application and the nature of the measurements and aggregation function.

* collector_hpke_config (HpkeConfig): The [HPKE] configuration of the Collector (described in Section 4.4.1); see Section 7 for information about the HPKE configuration algorithms.

* vdaf_verify_key (opaque byte string): 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 8.6.2.

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.

4.2.1. Batch Modes, Batches, and Queries

An aggregate result is computed from a set of reports, called a "batch". The Collector requests the aggregate result by making a "query" and the Aggregators use this query to select a batch for aggregation.

The task's batch mode defines both how reports are assigned into batches, how these batches are addressed and the semantics of the query used for collection. Regardless of batch mode, each report can only ever be part of a single batch.

Section 5 defines the time-interval and leader-selected batch modes and discusses how new batch modes may be defined by future documents.

The query is issued to the Leader by the Collector during the collection interaction (Section 4.6). Information used to guide batch selection is conveyed from the Leader to the Helper when initializing aggregation jobs (Section 4.5) and finalizing the aggregate shares.

4.3. Aggregation Parameter Validation

For each batch it collects, the Collector chooses an aggregation parameter used to verify the measurements before aggregating them. Before accepting a collection job from the Collector (Section 4.6.1), the Leader checks that the indicated aggregation parameter is valid according to the following procedure.

1. Decode the byte string agg_param into an AggParam as specified by the VDAF. If decoding fails, then the aggregation parameter is invalid.

2. Run vdaf.is_valid(decoded_agg_param, []), where decoded_agg_param is the decoded AggParam and is_valid() is as defined in Section 5.3 of [VDAF]. If the output is not True, then the aggregation parameter is invalid.

If both steps succeed, then the aggregation parameter is valid.

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 7 for information on HPKE algorithm choices.

Clients retrieve the HPKE configuration from each Aggregator by sending a GET to {aggregator}/hpke_config.

An Aggregator responds with an HpkeConfigList, with media type "application/ppm-dap;message=hpke-config-list". The HpkeConfigList contains one or more HpkeConfigs in decreasing order of preference. This allows an Aggregator to support multiple HPKE configurations and multiple sets of algorithms simultaneously.

```
HpkeConfig HpkeConfigList<10..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;
uint16 HpkeKemId;
uint16 HpkeKdfId;
```

The possible values for HpkeAeadId, HpkeKemId and HpkeKdfId are as defined in [HPKE], Section 7.

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

The Client MUST abort if:

* the response is not a valid HpkeConfigList;

* the HpkeConfigList is empty; or

* no HPKE config advertised by the Aggregator specifies a supported KEM, KDF and AEAD algorithm triple.

Aggregators SHOULD use caching to permit client-side caching of this resource [RFC9111]. Aggregators can control cache lifetime with the Cache-Control header, using a value appropriate to the lifetime of their keys. Aggregators SHOULD favor long cache lifetimes to avoid frequent cache revalidation, e.g., on the order of days.

Aggregators SHOULD continue to accept reports with old keys for at least twice the cache lifetime in order to avoid rejecting reports.

4.4.1.1. Example

```
GET /leader/hpke_config
Host: example.com

HTTP/1.1 200
Content-Type: application/ppm-dap;message=hpke-config-list
Cache-Control: max-age=86400

encoded([
  struct {
    id = 194,
    kem_id = 0x0010,
    kdf_id = 0x0001,
    aead_id = 0x0001,
    public_key = [0x01, 0x02, 0x03, 0x04, ...],
  } HpkeConfig,
  struct {
    id = 17,
    kem_id = 0x0020,
    kdf_id = 0x0001,
    aead_id = 0x0003,
    public_key = [0x04, 0x03, 0x02, 0x01, ...],
  } HpkeConfig,
])
```

4.4.2. Upload Request

Reports are uploaded using the reports resource, served by the Leader at {leader}/tasks/{task-id}/reports. An upload is represented as an UploadRequest, with media type "application/ppm-dap;message=upload-req", structured as follows:

```
struct {
  ReportID report_id;
  Time time;
  Extension public_extensions<0..2^16-1>;
} ReportMetadata;

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

struct {
  Report reports[message_length];
} UploadRequest;
```

Here message_length is the length of the HTTP message content ([RFC9110], Section 6.4).

Each upload request contains a sequence of Report with the following fields:

* report_metadata is public metadata describing the report.
  - report_id uniquely identifies the report. The Client MUST generate this by sampling 16 random bytes from a cryptographically secure random number generator.
  - time is the time at which the report was generated.
  - public_extensions is the list of public report extensions; see Section 4.4.3.
* public_share is the public share output by the VDAF sharding algorithm. 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 8.3). If it is used, client authentication MUST use a scheme that meets the requirements in Section 3.4.

4.4.2.1. Client Behavior

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(
    "dap-17" || task_id,
    measurement,
    report_id,
    rand,
)
```

* task_id is the task ID.

* measurement is the plaintext measurement, represented as the VDAF's Measurement associated type.

* report_id is the corresponding value from ReportMetadata, used as the nonce.

* rand is a random byte string of length specified by the VDAF. Each report's rand MUST be independently sampled from a cryptographically secure random number generator.

Vdaf.shard algorithm will return two input shares. The first is the Leader's input share, and the second is the Helper's.

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

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

* private_extensions is the list of private report extensions for the given Aggregator (see Section 4.4.3).

* payload is the Aggregator's input share.

Next, the Client encrypts each PlaintextInputShare as follows:

(RFC EDITOR: Once the document becomes an RFC, we will stop including the draft version in domain separation tags. In the remainder of this section, replace "dap-17" with "dap".)

```
enc, payload = SealBase(pk,
  "dap-17 input share" || 0x01 || server_role,
  input_share_aad, plaintext_input_share)
```

* pk is the public key from the Aggregator's HPKE configuration.

* 0x01 represents the Role of the sender (always the Client).

* server_role is the Role of the recipient (0x02 for the Leader and 0x03 for the Helper).

* plaintext_input_share is the Aggregator's PlaintextInputShare.

* input_share_aad is an encoded InputShareAad, constructed from the corresponding fields in the report per the definition below.

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

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

Clients upload reports by sending an UploadRequest as the body of a POST to the Leader's reports resource.

4.4.2.2. Leader Behavior

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

If the upload request is malformed, the Leader aborts with error invalidMessage.

If the Leader does not recognize the task ID, then it aborts with error unrecognizedTask.

If all the reports in the request are accepted, then the Leader sends a response with an empty body.

If some or all of the reports fail to upload for one of the reasons described in the remainder of this section, the Leader responds with a body consisting of an UploadErrors with the media type application/ppm-dap;message=upload-errors. The structure of the response is as follows:

```
struct {
  ReportID id;
  ReportError error;
} ReportUploadStatus;

struct {
  ReportUploadStatus status[message_length];
} UploadErrors;
```

Here message_length denotes the length in bytes of the concatenated ReportUploadStatus objects.

The Leader only includes reports that failed processing in the response. Reports that are accepted do not have a response.

Reports in the response MUST appear in the same order as in the request.

For each report that failed to upload, the Leader creates a ReportUploadStatus and includes the ReportId from the input and a ReportError (Section 4.1) that describes the failure. The length of this sequence is always less than or equal to the length of the upload sequence.

If the Leader does not recognize the config_id in the encrypted input share, it sets the corresponding error field to outdated_config. When the Client receives an outdated_config 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 discard it. In addition, it MAY set the corresponding error field to report_replayed.

The Leader MUST discard any report pertaining to a batch that has already been collected (see Section 2.3 for details). The Leader MAY also set the corresponding error field to report_replayed.

The Leader MUST discard any report whose timestamp is outside of the task's time_interval. When it does so, it SHOULD set the corresponding error field to report_dropped.

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 set the corresponding error field to report_too_early. In this situation, the Client MAY re-upload the report later on.

If the report contains an unrecognized public report extension, or if the Leader's input share contains an unrecognized private report extension, then the Leader MUST discard the report and MAY abort with error unsupportedExtension. If the Leader does abort for this reason, it SHOULD indicate the unsupported extensions in the resulting problem document via an extension member (Section 3.2 of [RFC9457]) unsupported_extensions on the problem document. This member MUST contain an array of numbers indicating the extension code points which were not recognized. For example, if the report upload contained two unsupported extensions with code points 23 and 42, the "unsupported_extensions" member would contain the JSON value [23, 42].

If the same extension type appears more than once among the public extensions and the private extensions in the Leader's input share, then the Leader MUST discard the report and MAY abort with error invalidMessage.

Validation of anti-replay and extensions is not mandatory during the handling of upload requests to avoid blocking on storage transactions or decryption of input shares. The Leader also cannot validate the Helper's extensions because it cannot decrypt the Helper's input share. Validation of report IDs and extensions will occur before aggregation.

4.4.2.3. Example

Successful upload

```
POST /leader/tasks/8BY0RzZMzxvA46_8ymhzycOB9krN-QIGYvg_RsByGec/reports
Host: example.com
Content-Type: application/ppm-dap;message=upload-req
Content-Length: 100

encoded(struct {
  reports = [
    struct {
      report_metadata = struct {
        report_id = [0x0a, 0x0b, 0x0c, 0x0d, ...],
        time = 17419860,
        public_extensions = [0x00, 0x00],
      } ReportMetadata,
      public_share = [0x0a, 0x0b, ...],
      leader_encrypted_input-share = struct {
        config_id = 1,
        enc = [0x0f, 0x0e, 0x0d, 0x0c, ...],
        payload = [0x0b, 0x0a, 0x09, 0x08, ...],
      } HpkeCiphertext,
      helper_encrypted_input-share = struct {
        config_id = 2,
        enc = [0x0c, 0x0d, 0x0e, 0x0f, ...],
        payload = [0x08, 0x00, 0x0a, 0x0b, ...],
      } HpkeCiphertext,
    } Report,
  ],
} UploadRequest)

HTTP/1.1 200

Failed upload of 1/2 reports submitted in one bulk upload

POST /leader/tasks/8BY0RzZMzxvA46_8ymhzycOB9krN-QIGYvg_RsByGec/reports
Host: example.com
Content-Type: application/ppm-dap;message=upload-req
Content-Length: 200

encoded(struct {
  reports = [
    struct {
      report_metadata = struct {
        report_id = [0x0a, 0x0b, 0x0c, 0x0d, ...],
        time = 20000000,
        public_extensions = [0x00, 0x01],
      } ReportMetadata,
      public_share = [0x0a, 0x0b, ...],
      leader_encrypted_input-share = struct {
        config_id = 1,
```

```
          enc = [0x0f, 0x0e, 0x0d, 0x0c, ...],
          payload = [0x0b, 0x0a, 0x09, 0x08, ...],
        } HpkeCiphertext,
        helper_encrypted_input-share = struct {
          config_id = 2,
          enc = [0x0c, 0x0d, 0x0e, 0x0f, ...],
          payload = [0x08, 0x00, 0x0a, 0x0b, ...],
        } HpkeCiphertext,
      } Report,
      struct {
        report_metadata = struct {
          report_id = [0x0z, 0x0y, 0x0x, 0x0w, ...],
          time = 20000000,
          public_extensions = [0x00, 0x01],
        } ReportMetadata,
        public_share = [0x0a, 0x0b, ...],
        leader_encrypted_input-share = struct {
          config_id = 1,
          enc = [0x0f, 0x0e, 0x0d, 0x0c, ...],
          payload = [0x0b, 0x0a, 0x09, 0x08, ...],
        } HpkeCiphertext,
        helper_encrypted_input-share = struct {
          config_id = 2,
          enc = [0x0c, 0x0d, 0x0e, 0x0f, ...],
          payload = [0x08, 0x00, 0x0a, 0x0b, ...],
        } HpkeCiphertext,
      } Report,
    ],
  } UploadRequest)

  HTTP/1.1 200
  Content-Type: application/ppm-dap;message=upload-errors
  Content-Length: 20

  encoded(struct {
    reports = [
      struct {
        id = [0x0z, 0x0y, 0x0x, 0x0w, ...],
        error = report_replayed,
      },
    ],
  } UploadErrors)
```

4.4.3. Report Extensions

Clients use report extensions to convey additional information to the Aggregators. Each ReportMetadata contains a list of extensions public to both aggregators, and each PlaintextInputShare contains a list of extensions private to the relevant Aggregator. For example, Clients may need to authenticate to the Helper by presenting a secret that must not be revealed to the Leader.

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

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

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

Field extension_type indicates the type of extension, and extension_data contains the opaque encoding of the extension.

Extensions are mandatory to implement. Unrecognized extensions are handled as specified in Section 4.5.2.4.

4.5. Verifying and Aggregating Reports

Once some 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 is parallelized across many "aggregation jobs" in which subsets of the reports are processed independently. Each aggregation job is associated with a single task, but a task can have many aggregation jobs.

An aggregation job runs the VDAF verification process described in [VDAF], Section 5.2 for each report in the job. Verification 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 a fixed operation depending only on the input share, but in general the mapping involves an aggregation parameter chosen by the Collector.

2. To verify that each pair of output shares, when combined, corresponds 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]) proves that the output shares sum up to an integer in a specific range, while the Prio3Histogram variant (Section 7.4.4 of [VDAF]) proves that output shares sum up to a one-hot vector representing a contribution to a single bucket of the histogram.

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 verification succeeds, then the resulting output shares are guaranteed to combine into a valid, refined measurement.

Aggregation jobs are identified by 16-byte job ID, chosen by the Leader:

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

An aggregation job is an HTTP resource served by the Helper at the URL {helper}/tasks/{task-id}/aggregation_jobs/{aggregation-job-id}. VDAF verification is mapped onto an aggregation job as illustrated in Figure 7. The first request from the Leader to the Helper includes the aggregation parameter, the Helper’s report share for each report in the job, and for each report the initialization step for verification. The Helper’s response, along with each subsequent request and response, carries the remaining messages exchanged during verification.

Figure 7: 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.

Each Aggregator maintains some state for each report. A state transition is triggered by receiving a message from the Aggregator's peer. Eventually this process results in a terminal state, either rejecting the report or recovering an output share. Once a report has reached a terminal state, no more messages will be processed for it. There are four possible states (see Section 5.7 of [VDAF]: Continued, FinishedWithOutbound, Finished and Rejected. The first two states include an outbound message to be processed by the peer.

The Helper can either process each step synchronously, meaning it computes each verification step before producing a response to the Leader's HTTP request, or asynchronously, meaning it responds immediately and defers processing to a background worker. To continue, the Leader polls the Helper until it responds with the next step. This choice allows a Helper implementation to choose a model that best fits its architecture and use case. For instance replay checks across vast numbers of reports and verification of large histograms, may be better suited for the asynchronous model.

Aggregation cannot begin until the Collector specifies a query and an aggregation parameter, except where eager aggregation (Section 4.5.1) is possible.

An aggregation job has three phases:

* Initialization: Disseminate report shares and initialize the VDAF verification state for each report.

* Continuation: Exchange verifier shares and messages until verification completes or an error occurs.

* Completion: Yield an output share for each report share in the aggregation job.

After an aggregation job is completed, each Aggregator commits to the output shares by updating running-total aggregate shares and other values for each batch bucket associated with a verified output share, as described in Section 4.5.3.3. These values are stored until a batch that includes the batch bucket is collected as described in Section 4.6.

The aggregation interaction provides protection against including reports in more than one batch and against adding reports to already collected batches, both of which can violate privacy (Section 2.3). Before committing to an output share, the Aggregators check whether its report ID has already been aggregated and whether the batch bucket being updated has been collected.

4.5.1. Eager Aggregation

In general, aggregation cannot begin until the Collector specifies a query and an aggregation parameter. However, depending on the VDAF and batch mode in use, it is often possible to begin aggregation as soon as reports arrive.

For example, Prio3 has just one valid aggregation parameter (the empty string), and so allows for eager aggregation. Both the time-interval and leader-selected batch modes defined in this document (Section 5) allow for eager aggregation, but future batch modes might preclude it.

Even when the VDAF uses a non-empty aggregation parameter, there still might be some applications in which the Aggregators can anticipate the parameter the Collector will choose and begin aggregation.

For example, when using Poplar1 (Section 8 of [VDAF]), the Collector and Aggregators might agree ahead of time on the set of candidate prefixes to use. In such cases, it is important that Aggregators ensure that the parameter eventually chosen by the Collector matches what they used. Depending on the VDAF, aggregating reports with multiple aggregation parameters may impact privacy. Aggregators must therefore ensure they only ever use the aggregation parameter chosen by the Collector.

4.5.2. Aggregate Initialization

Aggregation initialization accomplishes two tasks:

1. Determine which report shares are valid.

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

The Leader and Helper initialization behavior is detailed below.

4.5.2.1. Leader Initialization

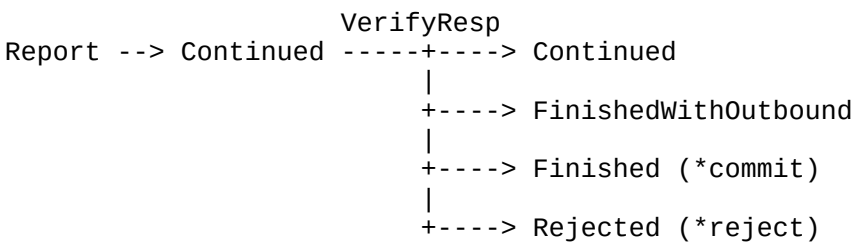

Figure 8: Leader state transition triggered by aggregation initialization. (*) indicates a terminal state.

The Leader begins an aggregation job by choosing a set of candidate reports that belong to the same task and a job ID which MUST be unique within the task.

First, the Leader MUST ensure each report in the candidate set can be committed per the criteria detailed in Section 4.5.3.3. If a report cannot be committed, then the Leader rejects it and removes it from the candidate set.

Next, the Leader decrypts each of its report shares as described in Section 4.5.2.3, then checks input share validity as described in Section 4.5.2.4. If either step fails, the Leader rejects the report and removes it from the candidate set.

For each report the Leader executes the following procedure:

```
state = Vdaf.ping_pong_leader_init(
    vdaf_verify_key,
    "dap-17" || task_id,
    agg_param,
    report_id,
    public_share,
    plaintext_input_share.payload,
)
```

where:

* vdaf_verify_key is the VDAF verification key for the task

* task_id is the task ID

* 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

ping_pong_leader_init is defined in Section 5.7.1 of [VDAF]. This process determines the initial per-report state. If state is of type Rejected (Section 5.7 of [VDAF], then the report is rejected and removed from the candidate set, and no message is sent to the Helper for this report.

Otherwise, if state is of type Continued (no other state is reachable at this point), then the state includes an outbound message denoted state.outbound. The Leader uses it to construct a VerifyInit structure for that report.

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

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

This message consists of:

* report_share.report_metadata: The report's metadata.

* report_share.public_share: The report's public share.

* report_share.encrypted_input_share: The Helper's encrypted input share.

* payload: The outbound message, set to state.outbound.

Once all the report shares have been initialized, the Leader creates an AggregationJobInitReq message containing the VerifyInit structures for the relevant reports.

```
struct {
  BatchMode batch_mode;
  opaque config<0..2^16-1>;
} PartialBatchSelector;

struct {
  opaque agg_param<0..2^32-1>;
  PartialBatchSelector part_batch_selector;
  VerifyInit verify_inits[verify_inits_length];
} AggregationJobInitReq;
```

This message consists of:

* agg_param: The VDAF aggregation parameter chosen by the Collector. Before initializing an aggregation job, the Leader MUST validate the parameter as described in Section 4.3.

* part_batch_selector: The "partial batch selector" used by the Aggregators to determine how to aggregate each report. Its contents depends on the indicated batch mode. This field is called the "partial" batch selector because depending on the batch mode, it may only partially determine a batch. See Section 5.

* verify_inits: the sequence of VerifyInit messages constructed in the previous step. Here verify_inits_length is the length of the HTTP message content ([RFC9110], Section 6.4), minus the lengths in octets of the encoded agg_param and part_batch_selector fields. That is, the remainder of the HTTP message consists of verify_inits.

The Leader sends the AggregationJobInitReq in the body of a PUT request to the aggregation job with a media type of "application/ppm-dap;message=aggregation-job-init-req". The Leader handles the response(s) as described in Section 3 to obtain an AggregationJobResp.

The AggregationJobResp.verify_resps field must include exactly the same report IDs in the same order as the Leader's AggregationJobInitReq. Otherwise, the Leader MUST abandon the aggregation job.

The Leader proceeds as follows with each report:

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

```
state = Vdaf.ping_pong_leader_continued(
    "dap-17" || task_id,
    agg_param,
    state,
    inbound,
)
```

where:

* task_id is the task ID

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

* state is the report's initial verification state

* inbound is the payload of the VerifyResp

If the new state has type Continued or FinishedWithOutbound, then there is at least one more outbound message to send before verification is complete. The Leader stores state and proceeds as in Section 4.5.3.1.

Else if the new state has type Finished, then verification is complete and the state includes an output share, denoted state.out_share. The Leader commits to state.out_share as described in Section 4.5.3.3.

Else if state has type Rejected, then the Leader rejects the report and removes it from the candidate set.

Note on rejection agreement: rejecting at this point would result in a batch mismatch if the Helper had already committed to its output share. This is impossible due to the verifiability property of the VDAF: if the underlying measurement were invalid, then the Helper would have indicated rejection in its response.

2. Else if the VerifyResp has type "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 report error is report_too_early, in which case the Leader MAY include the report in a subsequent aggregation job.

3. Otherwise the inbound message type is invalid for the Leader’s current state, in which case the Leader MUST abandon the aggregation job.

Since VDAF verification completes in a constant number of rounds, it will never be the case that verification is complete for some of the reports in an aggregation job but not others.

4.5.2.2. Helper Initialization

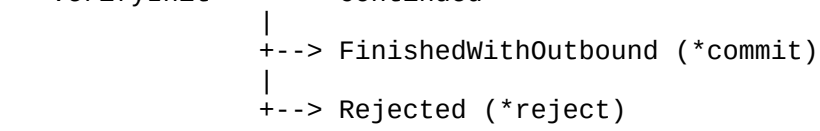

Figure 9: Helper state transition triggered by aggregation initialization. (*) indicates a terminal state.

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

The initialization request can be handled either asynchronously or synchronously as described in Section 3. When indicating that the job is not yet ready, the response MUST include a Location header field ([RFC9110], Section 10.2.2) set to the relative reference /tasks/{task-id}/aggregation_jobs/{aggregation-job-id}?step=0. Subsequent GET requests to the aggregation job MUST include the step query parameter so that the Helper can figure out which step of preparation the Leader is on (see Section 4.5.3.4). When the job is ready, the Helper responds with the AggregationJobResp (defined below).

Upon receipt of an AggregationJobInitReq, the Helper checks the following conditions:

* Whether it recognizes the task ID. If not, then the Helper MUST fail the job with error unrecognizedTask.

* Whether the AggregationJobInitReq is malformed. If so, the the Helper MUST fail the job with error invalidMessage.

* Whether the batch mode indicated by part_batch_selector.batch_mode matches the task's batch mode. If not, then the Helper MUST fail the job with error invalidMessage.

* Whether the aggregation parameter is valid as described in Section 4.3. If the aggregation parameter is invalid, then the Helper MUST fail the job with error invalidAggregationParameter.

* Whether the report IDs in AggregationJobInitReq.verify_inits are all distinct. If not, then the Helper MUST fail the job with error invalidMessage.

The Helper then processes the aggregation job by computing a response for each report share. This includes the following structures:

```
enum {
  continue(0),
  finish(1)
  reject(2),
  (255)
} VerifyRespType;

struct {
  ReportID report_id;
  VerifyRespType verify_resp_type;
  select (VerifyResp.verify_resp_type) {
    case continue: opaque payload<1..2^32-1>;
    case finish:   Empty;
    case reject:   ReportError report_error;
  };
} VerifyResp;
```

VerifyResp.report_id is always set to the ID of the report that the Helper is verifying. The values of the other fields in different cases are discussed below.

The Helper processes each of the remaining report shares in turn. First, the Helper decrypts each report share as described in Section 4.5.2.3, then checks input share validity as described in Section 4.5.2.4. If either decryption of validation fails, the Helper sets VerifyResp.verify_resp_type to reject and VerifyResp.report_error to the indicated error.

For all other reports it initializes the VDAF verification state as follows:

```
state = Vdaf.ping_pong_helper_init(
    vdaf_verify_key,
    "dap-17" || task_id,
    agg_param,
    report_id,
    public_share,
    plaintext_input_share.payload,
    inbound,
)
```

* vdaf_verify_key is the VDAF verification key for the task

* task_id is the task ID

* 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

* inbound is the payload of the inbound VerifyInit

This procedure determines the initial per-report state. If state is of type Rejected, then the Helper sets VerifyResp.verify_resp_type to reject and VerifyResp.report_error to vdaf_verify_error.

Otherwise state has type Continued or FinishedWithOutbound and there is at least one more outbound message to process. State Finished is not reachable at this point. The Helper sets VerifyResp.verify_resp_type to continue and VerifyResp.payload to state.outbound.

If state has type Continued, then the Helper stores state for use in the first continuation step in Section 4.5.3.2.

Else if state has type FinishedWithOutbound, then the Helper commits to state.out_share as described in Section 4.5.3.3. If commitment fails with some report error commit_error (e.g., the report was replayed or its batch bucket was collected), then the Helper sets VerifyResp.verify_resp_type to reject and VerifyResp.report_error to commit_error.

Once the Helper has constructed a VerifyResp for each report, the aggregation job response is ready. Its results are represented by an AggregationJobResp, which is structured as follows:

```
struct {
  VerifyResp verify_resps[message_length];
} AggregationJobResp;
```

Here message_length is the length of the HTTP message content ([RFC9110], Section 6.4).

verify_resps is the outbound VerifyResp messages for each report computed in the previous step. The order MUST match AggregationJobInitReq.verify_inits. The media type for AggregationJobResp is "application/ppm-dap;message=aggregation-job-resp".

The Helper may receive multiple copies of a given initialization request. The Helper MUST verify that subsequent requests have the same AggregationJobInitReq value and abort with a client error if

they do not. It is illegal to rewind or reset the state of an aggregation job. If the Helper receives requests to initialize an aggregation job once it has been continued at least once, confirming that the Leader received the Helper's response (see Section 4.5.3), it MUST abort with a client error.

4.5.2.3. Input Share Decryption

Each report share has a corresponding task ID, report metadata (report ID, timestamp, and public extensions), 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 attempts decryption of the payload with the following procedure:

```
plaintext_input_share = OpenBase(encrypted_input_share.enc, sk,
  "dap-17 input share" || 0x01 || server_role,
  input_share_aad, encrypted_input_share.payload)
```

* sk is the secret key from the HPKE configuration indicated by encrypted_input_share.config_id

* 0x01 represents the Role of the sender (always the Client)

* 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 the HPKE configuration ID is unrecognized or 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.

4.5.2.4. Input Share Validation

Before initialization, Aggregators MUST perform the following checks for each input share in the job, in any order:

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’s timestamp is more than a few minutes ahead of the current time. If so, then the Aggregator SHOULD mark the input share as invalid with error report_too_early.

3. Check if the report’s timestamp is before the task’s task_interval. If so, the Aggregator MUST mark the input share as invalid with the error task_not_started.

4. Check if the report’s timestamp is after the task’s task_interval. If so, the Aggregator MUST mark the input share as invalid with the error task_expired.

5. Check if the public or private report extensions contain any unrecognized report extension types. If so, the Aggregator MUST mark the input share as invalid with error invalid_message.

6. Check if any two extensions have the same extension type across public and private extension fields. If so, the Aggregator MUST mark the input share as invalid with error invalid_message.

7. If an Aggregator cannot determine if an input share is valid--for example, the report timestamp may be so far in the past that the state required to perform the check has been evicted from the Aggregator’s storage (see Section 6.4.1 for details)--it MUST mark the input share as invalid with error report_dropped.

If all of the above checks succeed, the input share is valid.

4.5.2.5. Example

The Helper handles the aggregation job initialization synchronously:

```
PUT /helper/tasks/8BY0RzZMzxvA46_8ymhzycOB9krN-QIGYvg_RsByGec/\
  aggregation_jobs/lc7aUeGpdSNosNlh-UZhKA
Host: example.com
Content-Type: application/ppm-dap;message=aggregation-job-init-req
Content-Length: 100
Authorization: Bearer auth-token

encoded(struct {
  agg_param = [0x00, 0x01, 0x02, 0x04, ...],
  part_batch_selector = struct {
    batch_mode = BatchMode.leader_selected,
    config = encoded(struct {
      batch_id = [0x1f, 0x1e, ..., 0x00],
    } LeaderSelectedPartialBatchSelectorConfig),
  } PartialBatchSelector,
  verify_inits,
} AggregationJobInitReq)

HTTP/1.1 200
Content-Type: application/ppm-dap;message=aggregation-job-resp
Content-Length: 100

encoded(struct { verify_resps } AggregationJobResp)
```

Or asynchronously:

```
PUT /helper/tasks/8BY0RzZMzxvA46_8ymhzycOB9krN-QIGYvg_RsByGec/\
  aggregation_jobs/lc7aUeGpdSNosNlh-UZhKA
Host: example.com
Content-Type: application/ppm-dap;message=aggregation-job-init-req
Content-Length: 100
Authorization: Bearer auth-token

encoded(struct {
  agg_param = [0x00, 0x01, 0x02, 0x04, ...],
  part_batch_selector = struct {
    batch_mode = BatchMode.time_interval,
    config = encoded(Empty),
  },
  verify_inits,
} AggregationJobInitReq)

HTTP/1.1 200
Location: /helper/tasks/8BY0RzZMzxvA46_8ymhzycOB9krN-QIGYvg_RsByGec/\
  aggregation_jobs/lc7aUeGpdSNosNlh-UZhKA?step=0
Retry-After: 300

GET /helper/tasks/8BY0RzZMzxvA46_8ymhzycOB9krN-QIGYvg_RsByGec/\
  aggregation_jobs/lc7aUeGpdSNosNlh-UZhKA?step=0
Host: example.com
Authorization: Bearer auth-token

HTTP/1.1 200
Location: /helper/tasks/8BY0RzZMzxvA46_8ymhzycOB9krN-QIGYvg_RsByGec/\
  aggregation_jobs/lc7aUeGpdSNosNlh-UZhKA?step=0
Retry-After: 300

GET /helper/tasks/8BY0RzZMzxvA46_8ymhzycOB9krN-QIGYvg_RsByGec/\
  aggregation_jobs/lc7aUeGpdSNosNlh-UZhKA?step=0
Host: example.com
Authorization: Bearer auth-token

HTTP/1.1 200
Content-Type: application/ppm-dap;message=aggregation-job-resp
Content-Length: 100

encoded(struct { verify_resps } AggregationJobResp)
```

4.5.3. Aggregate Continuation

In the continuation phase, the Leader drives the VDAF verification of each report in the candidate set until the underlying VDAF moves into a terminal state, yielding an output share for each Aggregator or a rejection.

Continuation is only required for VDAFs that require more than one round. Single round VDAFs like Prio3 will never reach this phase.

4.5.3.1. Leader Continuation

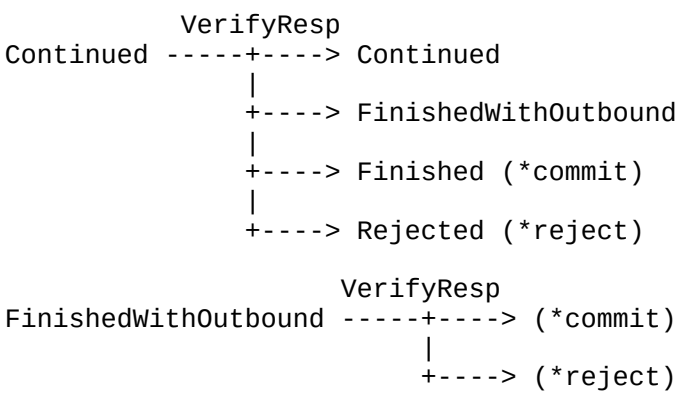

Figure 10: Leader state transition triggered by aggregation continuation. (*) indicates a terminal state.

The Leader begins each step of aggregation continuation with a verification state object state for each report in the candidate set. Either all states have type Continued or all states have type FinishedWithOutbound. In either case, there is at least one more outbound message to process, denoted state.outbound.

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 verification continuation message:

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

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

Next, the Leader sends a POST to the aggregation job with media type "application/ppm-dap;message=aggregation-job-continue-req" and body structured as:

```
struct {
  uint16 step;
  VerifyContinue verify_continues[verify_continues_length];
} AggregationJobContinueReq;
```

The step field is the step of DAP aggregation that the Leader just reached and wants the Helper to advance to. The verify_continues field is the sequence of verification continuation messages constructed in the previous step. Here verify_continues_length is the length of the HTTP message content ([RFC9110], Section 6.4), minus the length in octets of step. The VerifyContinue elements MUST be in the same order as the previous request to the aggregation job, omitting any reports that were previously rejected by either Aggregator.

The Leader handles the response(s) as described in Section 3 to obtain an AggregationJobResp.

The response's verify_resps MUST include exactly the same report IDs in the same order as the Leader's AggregationJobContinueReq. Otherwise, the Leader MUST abandon the aggregation job.

Otherwise, the Leader proceeds as follows with each report:

1. If state has type Continued and the inbound VerifyResp has type "continue", then the Leader computes

```
state = Vdaf.ping_pong_leader_continued(
    "dap-17" || task_id,
    agg_param,
    state,
    inbound,
)
```

where task_id is the task ID, inbound is the payload of the inbound VerifyResp, and state is the report's verification state carried over from the previous step. It then processes the next state transition:

* If the type of the new state is Continued or FinishedWithOutbound, then the Leader stores state and proceeds to the next continuation step.

* Else if the new type is Rejected, then the Leader rejects the report and removes it from the candidate set.

Note on rejection agreement: rejecting at this point would result in a batch mismatch if the Helper had already committed to its output share. This is impossible due to the verifiability property of the VDAF: if the underlying measurement were invalid, then the Helper would have indicated rejection in its response.

* Else if the new type is Finished, then the Leader commits to state.out_share as described in Section 4.5.3.3.

2. Else if state has type FinishedWithOutbound and the inbound VerifyResp has type "finish", then verification is complete and the Leader commits to state.out_share as described in Section 4.5.3.3.

3. Else if the inbound VerifyResp has type "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 report error is report_too_early, in which case the Leader MAY include the report in a subsequent aggregation job.

4. Otherwise the inbound message is incompatible with the Leader's current state, in which case the Leader MUST abandon the aggregation job.

If the Leader fails to process the response from the Helper, for example because of a transient failure such as a network connection failure or process crash, the Leader SHOULD re-send the original request unmodified in order to attempt recovery (see Section 4.5.3.4).

4.5.3.2. Helper Continuation

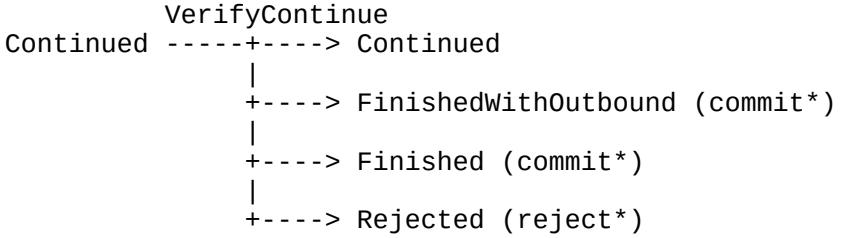

Figure 11: Helper state transition triggered by aggregation continuation. (*) indicates a terminal state.

The Helper begins continuation with a state object for each report in the candidate set, each of which has type Continued. The Helper waits for the Leader to POST an AggregationJobContinueReq to the aggregation job.

The continuation request can be handled either asynchronously or synchronously as described in Section 3. When indicating that the job is not yet ready, the response MUST include a Location header field ([RFC9110], Section 10.2.2) to the relative reference /tasks/{task-id}/aggregation_jobs/{aggregation-job-id}?step={step}, where step is set to AggregationJobContinueReq.step. Subsequent GET requests to the aggregation job MUST include the step query parameter so that the Helper can figure out which step of preparation the Leader is on (see Section 4.5.3.4). The representation of the aggregation job is an AggregationJobResp.

To begin handling an AggregationJobContinueReq, the Helper checks the following conditions:

* Whether it recognizes the task ID. If not, then the Helper MUST fail the job with error unrecognizedTask.

* Whether it recognizes the indicated aggregation job ID. If not, the Helper MUST fail the job with error unrecognizedAggregationJob.

* Whether the AggregationJobContinueReq is malformed. If so, the the Helper MUST fail the job with error invalidMessage.

* Whether AggregationJobContinueReq.step is equal to 0. If so, the Helper MUST fail the job with error invalidMessage.

* Whether the report IDs are all distinct and each report ID corresponds to one of the state objects. If either of these checks fail, then the Helper MUST fail the job with error invalidMessage.

Additionally, if any verification step appears out of order relative to the previous request, then the Helper MAY fail the job with error invalidMessage. A report may be missing, in which case the Helper assumes the Leader rejected it and removes it from the candidate set.

Next, the Helper checks the continuation step indicated by the request. If the step value is one greater than the job’s current step, then the Helper proceeds.

The Helper may receive multiple copies of a continuation request for a given step. The Helper MAY attempt to recover by sending the same response as it did for the previous AggregationJobContinueReq, without performing any additional work on the aggregation job. It is illegal to rewind or reset the state of an aggregation job, so in this case it MUST verify that the contents of the AggregationJobContinueReq are identical to the previous message (see Section 4.5.3.4).

If the Helper does not wish to attempt recovery, or if the step has some other value, the Helper MUST fail the job with error stepMismatch.

For each report, the Helper does the following:

```
state = Vdaf.ping_pong_helper_continued(
    "dap-17" || task_id,
    agg_param,
    state,
    inbound,
)
```

where task_id is the task ID, inbound is the payload of the inbound VerifyContinue, and state is the report's verification state carried over from the previous step. If the new state has type Rejected, then the Helper sets VerifyResp.verify_resp_type to reject and VerifyResp.report_error to vdaf_verify_error.

If state has type Continued, then the Helper stores state for use in the next continuation step.

If state has type FinishedWithOutbound or Finished, then the Helper commits to state.out_share as described in Section 4.5.3.3. If commitment fails with some report error commit_error (e.g., the report was replayed or its batch bucket was collected), then the Helper sets VerifyResp.verify_resp_type to reject and VerifyResp.report_error to commit_error.

If commitment succeeds, the Helper's response depends on whether the state includes an outbound message that needs to be processed. If state has type Continued or FinishedWithOutbound then the Helper sets VerifyResp.verify_resp_type to continue and VerifyResp.payload to state.outbound.

Otherwise, if state has type Finished, then the Helper sets VerifyResp.verify_resp_type to finish.

Once the Helper has computed a VerifyResp for every report, the aggregation job response is ready. It is represented by an AggregationJobResp message (see Section 4.5.2.2) with each verification step. The order of the verification steps MUST match the Leader's AggregationJobContinueReq.

4.5.3.3. Batch Buckets

When aggregation refines an output share, it must be stored into an appropriate "batch bucket", which is defined in this section. An output share cannot be removed from a batch bucket once stored, so we say that the Aggregator _commits_ the output share. The data stored in a batch bucket is kept for eventual use in the Section 4.6.

Batch buckets are indexed by a "batch bucket identifier" as as specified by the task's batch mode:

* For the time-interval batch mode (Section 5.1, the batch bucket identifier is an interval of time and is determined by the report's timestamp.

* For the leader-selected batch mode (Section 5.2), the batch bucket identifier is the batch ID and indicated in the aggregation job.

A few different pieces of information are associated with each batch bucket:

* An aggregate share, as defined by the [VDAF] in use.

* The number of reports included in the batch bucket.

* A 32-byte checksum value, as defined below.

An output share is eligible to be committed if the following conditions are met:

* If the batch bucket has been collected, the Aggregator MUST mark the report invalid with error batch_collected.

* If the report ID associated with out_share has been aggregated in the task, the Aggregator MUST mark the report invalid with error report_replayed.

Aggregators MUST check these conditions before committing an output share. Helpers may perform these checks at any time before commitment (i.e., during either aggregation initialization or continuation), but Leaders MUST perform these checks before adding a report to an aggregation job.

The following procedure is used to commit an output share out_share to a batch bucket:

* Look up the existing batch bucket for the batch bucket identifier associated with the aggregation job and output share.

  - If there is no existing batch bucket, initialize a new one. The initial aggregate share value is computed as Vdaf.agg_init(agg_param), where agg_param is the aggregation parameter associated with the aggregation job (see [VDAF], Section 4.4). The initial count is 0 and the initial checksum is 32 zero bytes.

* Update the aggregate share agg_share to Vdaf.agg_update(agg_param, agg_share, out_share).

* Increment the count by 1.

* Update the checksum value to the bitwise XOR of the current checksum value with the SHA256 [SHS] hash of the report ID associated with the output share.

* Store out_share's report ID for future replay checks.

This section describes a single set of values associated with each batch bucket. However, implementations are free to shard batch buckets, combining them back into a single set of values when reading the batch bucket. The aggregate shares are combined using Vdaf.merge(agg_param, agg_shares) (see [VDAF], Section 4.4), the count values are combined by summing, and the checksum values are combined by bitwise XOR.

Implementation note: The Leader considers a batch to be collected once it has completed a collection job for a CollectionJobReq 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 conveyed in these messages. Queries must satisfy the criteria defined by their batch mode (Section 5). These criteria are meant to restrict queries in a way that makes it easy to determine whether a report pertains to a batch that was collected. See Section 6.4.2 for more information.

4.5.3.4. Recovering from Aggregation Step Skew

AggregationJobContinueReq messages contain a step field, allowing Aggregators to ensure that their peer is on an expected step of verification. 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 AggregationJobResp that was previously dropped. To make that kind of recovery possible, Aggregator implementations SHOULD checkpoint the most recent step's verification 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 aggregation step skew recovery, 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.

One way the Helper could achieve 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).

4.5.3.5. Example

The Helper handles the aggregation job continuation synchronously:

```
POST /helper/tasks/8BY0RzZMzxvA46_8ymhzycOB9krN-QIGYvg_RsByGec/\
  aggregation_jobs/lc7aUeGpdSNosNlh-UZhKA
Host: example.com
Content-Type: application/ppm-dap;message=aggregation-job-continue-req
Content-Length: 100
Authorization: Bearer auth-token

encoded(struct {
  step = 1,
  verify_continues,
} AggregationJobContinueReq)

HTTP/1.1 200
Content-Type: application/ppm-dap;message=aggregation-job-resp
Content-Length: 100

encoded(struct { verify_resps } AggregationJobResp)
```

Or asynchronously:

```
POST /helper/tasks/8BY0RzZMzxvA46_8ymhzycOB9krN-QIGYvg_RsByGec/\
  aggregation_jobs/lc7aUeGpdSNosNlh-UZhKA
Host: example.com
Content-Type: application/ppm-dap;message=aggregation-job-continue-req
Content-Length: 100
Authorization: Bearer auth-token

encoded(struct {
  step = 1,
  verify_continues,
} AggregationJobContinueReq)

HTTP/1.1 200
Location: /helper/tasks/8BY0RzZMzxvA46_8ymhzycOB9krN-QIGYvg_RsByGec/\
  aggregation_jobs/lc7aUeGpdSNosNlh-UZhKA?step=1
Retry-After: 300

GET /helper/tasks/8BY0RzZMzxvA46_8ymhzycOB9krN-QIGYvg_RsByGec/\
  aggregation_jobs/lc7aUeGpdSNosNlh-UZhKA?step=1
Host: example.com
Authorization: Bearer auth-token

HTTP/1.1 200
Location: /helper/tasks/8BY0RzZMzxvA46_8ymhzycOB9krN-QIGYvg_RsByGec/\
  aggregation_jobs/lc7aUeGpdSNosNlh-UZhKA?step=1
Retry-After: 300

GET /helper/tasks/8BY0RzZMzxvA46_8ymhzycOB9krN-QIGYvg_RsByGec/\
  aggregation_jobs/lc7aUeGpdSNosNlh-UZhKA?step=1
Host: example.com
Authorization: Bearer auth-token

HTTP/1.1 200
Content-Type: application/ppm-dap;message=aggregation-job-resp
Content-Length: 100

encoded(struct { verify_resps } AggregationJobResp)
```

4.5.4. Aggregation Job Abandonment and Deletion

This document describes various error cases where a Leader is required to abandon an aggregation job. This means the Leader should discontinue further processing of the job and the reports included in it. Reports included in an abandoned aggregation job MUST NOT be considered collected for purposes of replay and double collection checks (Section 4.5.3.3) and MAY be included in future aggregation

jobs.

If the Leader must abandon an aggregation job, it SHOULD let the Helper know it can clean up its state by sending a DELETE request to the job. Deletion of a completed aggregation job MUST NOT delete information needed for replay or double collection checks.

4.5.4.1. Example

```
DELETE /helper/tasks/8BY0RzZMzxvA46_8ymhzycOB9krN-QIGYvg_RsByGec/\
  aggregation_jobs/lc7aUeGpdSNosNlh-UZhKA
Host: example.com
Authorization: Bearer auth-token

HTTP/1.1 200
```

4.6. Collecting Results

The Collector initiates this phase with a query to the Leader (Section 5), 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 process is referred to as a "collection job" and 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 4.6.3

Once complete, the Collector computes the final aggregate result as specified in Section 4.6.5.

Collection jobs are identified by a 16-byte job ID, chosen by the Collector:

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

A collection job is an HTTP resource served by the Leader at the URL {leader}/tasks/{task-id}/collection_jobs/{collection-job-id}.

4.6.1. Collection Job Initialization

First, the Collector chooses a collection job ID, which MUST be unique within the scope of the corresponding DAP task.

To initiate the collection job, the Collector issues a PUT request to the collection job with media type "application/ppm-dap;message=collection-job-req", and a body structured as follows:

```
struct {
  BatchMode batch_mode;
  opaque config<0..2^16-1>;
} Query;

struct {
  Query query;
  opaque agg_param<0..2^32-1>;
} CollectionJobReq;
```

* query, the Collector's query. The content of this field depends on the indicated batch mode (Section 5).

* agg_param, an aggregation parameter for the VDAF being executed. This is the same value as in AggregationJobInitReq (see Section 4.5.2.1).

Depending on the VDAF scheme and how the Leader is configured, the Leader and Helper may already have aggregated 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 (see Section 4.5.1), 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.

The collection request can be handled either asynchronously or synchronously as described in Section 3. The representation of the collection job is a CollectionJobResp (defined below).

If the job fails with invalidBatchSize, then the Collector MAY retry it later, once it believes enough new reports have been uploaded and aggregated to allow the collection job to succeed.

The Leader begins handling a CollectionJobReq by checking the following conditions:

* Whether it recognizes the task ID. If not, the Leader MUST fail the collection job with error unrecognizedTask.

* Whether the indicated batch mode matches the task's batch mode. If not, the Leader MUST fail the job with error invalidMessage.

* Whether the CollectionJobReq is malformed. If so, the the Helper MUST fail the job with error invalidMessage.

* Whether the aggregation parameter is valid as described in Section 4.3. If not, the Leader MUST fail the job with error invalidAggregationParameter.

* Whether the Query in the Collector's request determines a batch that can be collected. If the query does not identify a valid set of batch buckets according to the criteria defined by the batch mode in use (Section 5), then the Leader MUST fail the job with error batchInvalid.

* Whether any of the batch buckets identified by the query have already been collected, then the Leader MUST fail the job with error batchOverlap.

* If aggregation was performed eagerly (Section 4.5.1), then the Leader checks that the aggregation parameter received in the CollectionJobReq matches the aggregation parameter used in each aggregation job pertaining to the batch. If not, the Leader MUST fail the job with error invalidMessage.

Having validated the CollectionJobReq, the Leader begins working with the Helper to aggregate the reports satisfying the query (or continues this process, depending on whether the Leader is aggregating eagerly; Section 4.5.1) as described in Section 4.5.

If the Leader has a pending aggregation job that overlaps with the batch 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 batch mismatch errors.

If the number of validated reports in the batch is not equal to or greater than the task's minimum batch size, then the Leader SHOULD wait for more reports to be uploaded and aggregated and try the collection job again later. Alternately, the Leader MAY give up on the collection job (for example, if it decides that no new reports satisfying the query are likely to ever arrive), in which case it MUST fail the job with error invalidBatchSize. It MUST NOT fulfill any jobs with an insufficient number of validated reports.

Once the Leader has validated the collection job and run to completion all the aggregation jobs that pertain to it, it obtains the Helper's aggregate share following the aggregate-share request flow described in Section 4.6.3. If obtaining the aggregate share fails, then the Leader MUST fail the collection job with the error that caused the failure.

Once the Leader has the Helper's aggregate share and has computed its own, the collection job is ready. Its results are represented by a CollectionJobResp, which is structured as follows:

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

A CollectionJobResp's media type is "application/ppm-dap;message=collection-job-resp". The structure includes the following:

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

* 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. In the case of a time-interval query (Section 5.1), this interval can be smaller than the one in the corresponding CollectionJobReq.query.

* leader_encrypted_agg_share: The Leader's aggregate share, encrypted to the Collector (see Section 4.6.6).

* helper_encrypted_agg_share: The Helper's aggregate share, encrypted to the Collector (see Section 4.6.6).

Once the Leader has constructed a CollectionJobResp for the Collector, the Leader considers the batch to be collected, and further aggregation jobs MUST NOT commit more reports to the batch (see Section 4.5.3.3).

Changing a collection job's parameters is illegal, so if there are further PUT requests to the collection job with a different CollectionJobReq, the Leader MUST abort with error invalidMessage.

4.6.1.1. Example

The Leader handles the collection job request synchronously:

```
PUT /leader/tasks/8BY0RzZMzxvA46_8ymhzycOB9krN-QIGYvg_RsByGec/\
  collection_jobs/lc7aUeGpdSNosNlh-UZhKA
Host: example.com
Content-Type: application/ppm-dap;message=collection-job-req
Authorization: Bearer auth-token

encoded(struct {
  query = struct {
    batch_mode = BatchMode.leader_selected,
    query = encoded(Empty),
  } Query,
  agg_param = [0x00, 0x01, ...],
} CollectionJobReq)

HTTP/1.1 200
Content-Type: application/ppm-dap;message=collection-job-resp

encoded(struct {
  part_batch_selector = struct {
    batch_mode = BatchMode.leader_selected,
    config = encoded(struct {
      batch_id = [0x1f, 0x1e, ..., 0x00],
    } LeaderSelectedPartialBatchSelectorConfig),
  } PartialBatchSelector,
  report_count = 1000,
  interval = struct {
    start = 16595440,
    duration = 1,
  } Interval,
  leader_encrypted_agg_share = struct { ... } HpkeCiphertext,
  helper_encrypted_agg_share = struct { ... } HpkeCiphertext,
} CollectionJobResp)
```

Or asynchronously:

```
PUT /leader/tasks/8BY0RzZMzxvA46_8ymhzycOB9krN-QIGYvg_RsByGec/\
  collection_jobs/lc7aUeGpdSNosNlh-UZhKA
Host: example.com
Content-Type: application/ppm-dap;message=collection-job-req
Authorization: Bearer auth-token

encoded(struct {
  query = struct {
    batch_mode = BatchMode.time_interval,
    query = encoded(struct {
      batch_interval = struct {
        start = 16595440,
        duration = 1,
      } Interval,
    } TimeIntervalQueryConfig),
  },
  agg_param = encoded(Empty),
} CollectionJobReq)

HTTP/1.1 200
Retry-After: 300

GET /leader/tasks/8BY0RzZMzxvA46_8ymhzycOB9krN-QIGYvg_RsByGec/\
  collection_jobs/lc7aUeGpdSNosNlh-UZhKA
Host: example.com
Authorization: Bearer auth-token

HTTP/1.1 200
Retry-After: 300

GET /leader/tasks/8BY0RzZMzxvA46_8ymhzycOB9krN-QIGYvg_RsByGec/\
  collection_jobs/lc7aUeGpdSNosNlh-UZhKA
Host: example.com
Authorization: Bearer auth-token

HTTP/1.1 200
Content-Type: application/ppm-dap;message=collection-job-resp

encoded(struct {
  part_batch_selector = struct {
    batch_mode = BatchMode.time_interval,
    config = encoded(struct {
      interval = struct {
        start = 1659544,
        duration = 10,
      } Interval,
    } TimeIntervalBatchSelectorConfig)
  },
```

```
    report_count = 4000,
    interval = struct {
      start = 1659547,
      duration = 10,
    } Interval,
    leader_encrypted_agg_share = struct { ... } HpkeCiphertext,
    helper_encrypted_agg_share = struct { ... } HpkeCiphertext,
  } CollectionJobResp)
```

4.6.2. Collection Job Deletion

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

Aggregators MUST NOT delete information needed for replay or double collection checks (Section 4.5.3.3).

4.6.2.1. Example

```
DELETE /leader/tasks/8BY0RzZMzxvA46_8ymhzycOB9krN-QIGYvg_RsByGec/\
  collection_jobs/lc7aUeGpdSNosNlh-UZhKA
Host: example.com
Authorization: Bearer auth-token

HTTP/1.1 200
```

4.6.3. Obtaining Aggregate Shares

The Leader must compute its own aggregate share and obtain the Helper's encrypted aggregate share before it can complete a collection job.

First, the Leader retrieves all batch buckets (Section 4.5.3.3) associated with this collection job. The batch buckets to retrieve depend on the batch mode of this task:

* For time-interval (Section 5.1), this is all batch buckets whose batch bucket identifiers are contained within the batch interval specified in the CollectionJobReq's query.

* For leader-selected (Section 5.2), this is the batch bucket associated with the batch ID the Leader has chosen for this collection job.

The Leader then combines the values inside the batch bucket as follows:

* Aggregate shares are combined via Vdaf.merge(agg_param, agg_shares) (see [VDAF], Section 4.4), where agg_param is the aggregation parameter provided in the CollectionJobReq, and agg_shares are the (partial) aggregate shares in the batch buckets. The result is the Leader aggregate share for this collection job.

* Report counts are combined via summing.

* Checksums are combined via bitwise XOR.

A Helper aggregate share is identified by a 16-byte ID:

```
opaque AggregateShareID[16];
```

The Helper's aggregate share is an HTTP resource served by the Helper at the URL {helper}/tasks/{task-id}/aggregate_shares/{aggregate-share-id}. To obtain it, the Leader first chooses an aggregate share ID, which MUST be unique within the scope of the corresponding DAP task.

Then the Leader sends a PUT request to the aggregate share with the body:

```
struct {
  BatchMode batch_mode;
  opaque config<0..2^16-1>;
} BatchSelector;

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

The media type of AggregateShareReq is "application/ppm-dap;message=aggregate-share-req". The structure contains the following parameters:

* batch_selector: The "batch selector", the contents of which depends on the indicated batch mode (see Section 5.

* agg_param: The encoded aggregation parameter for the VDAF being executed.

* report_count: The number number of reports included in the batch, as computed above.

* checksum: The batch checksum, as computed above.

The aggregate share request can be handled either asynchronously or synchronously as described in Section 3. The representation of the share is an AggregateShare (defined below).

The Helper first ensures that it recognizes the task ID. If not, it MUST fail the job with error unrecognizedTask.

The indicated batch mode MUST match the task's batch mode. If not, the Helper MUST fail the job with error invalidMessage.

If the AggregateShareReq is malformed, the Helper MUST fail the job with error invalidMessage.

The Helper then verifies that the BatchSelector in the Leader's request determines a batch that can be collected. If the selector does not identify a valid set of batch buckets according to the criteria defined by the batch mode in use (Section 5), then the Helper MUST fail the job with error batchInvalid.

If any of the batch buckets identified by the selector have already been collected, then the Helper MUST fail the job with error batchOverlap.

If the number of validated reports in the batch is not equal to or greater than the task's minimum batch size, then the Helper MUST abort with invalidBatchSize.

The aggregation parameter MUST match the aggregation parameter used in aggregation jobs pertaining to this batch. If not, the Helper MUST fail the job with error invalidMessage.

Next, the Helper retrieves and combines the batch buckets associated with the request using the same process used by the Leader (described at the beginning of this section), arriving at its aggregate share, report count, and checksum values. If the Helper's computed report count and checksum values do not match the values provided in the AggregateShareReq, it MUST fail the job with error batchMismatch.

The Helper then encrypts agg_share under the Collector's HPKE public key as described in Section 4.6.6, 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.

Once the Helper has encrypted its aggregate share, the aggregate share job is ready. Its results are represented by an AggregateShare, with media type "application/ppm-dap;message=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.

After receiving the Helper's response, the Leader includes the HpkeCiphertext in its response to the Collector (see Section 4.6.5).

Once an AggregateShareReq has been constructed for the batch determined by a given query, the Helper considers the batch to be collected. The Helper MUST NOT commit any more output shares to the batch. It is an error for the Leader to issue any more aggregation jobs for additional reports that satisfy the query. These reports MUST be rejected by the Helper as described in Section 4.5.3.3.

Changing an aggregate share's parameters is illegal, so if there are further PUT requests to the aggregate share with a different AggregateShareReq, the Helper MUST abort with error invalidMessage.

Before completing the collection job, the Leader encrypts its aggregate share under the Collector's HPKE public key as described in Section 4.6.6.

4.6.3.1. Example

The Helper handles the aggregate share request synchronously:

```
PUT /helper/tasks/8BY0RzZMzxvA46_8ymhzycOB9krN-QIGYvg_RsByGec/\
  aggregate_shares/lc7aUeGpdSNosNlh-UZhKA
Host: example.com
Content-Type: application/ppm-dap;message=aggregate-share-req
Authorization: Bearer auth-token

encoded(struct {
  batch_selector = struct {
    batch_mode = BatchMode.time_interval,
    config = encoded(struct {
      batch_interval = struct {
        start = 1659544,
        duration = 10,
      } Interval,
    } TimeIntervalBatchSelectorConfig),
  } BatchSelector,
  agg_param = [0x00, 0x01, ...],
  report_count = 1000,
  checksum = [0x0a, 0x0b, ..., 0x0f],
} AggregateShareReq)

HTTP/1.1 200
Content-Type: application/ppm-dap;message=aggregate-share

encoded(struct {
  encrypted_aggregate_share = struct { ... } HpkeCiphertext,
} AggregateShare)
```

Or asynchronously:

```
PUT /helper/tasks/8BY0RzZMzxvA46_8ymhzycOB9krN-QIGYvg_RsByGec/\
  aggregate_shares/lc7aUeGpdSNosNlh-UZhKA
Host: example.com
Content-Type: application/ppm-dap;message=aggregate-share-req
Authorization: Bearer auth-token

encoded(struct {
  batch_selector = struct {
    batch_mode = BatchMode.time_interval,
    config = encoded(struct {
      batch_interval = struct {
        start = 1659544,
        duration = 10,
      } Interval,
    } TimeIntervalBatchSelectorConfig),
  } BatchSelector,
  agg_param = [0x00, 0x01, ...],
  report_count = 1000,
  checksum = [0x0a, 0x0b, ..., 0x0f],
} AggregateShareReq)

HTTP/1.1 200
Retry-After: 300

GET /helper/tasks/8BY0RzZMzxvA46_8ymhzycOB9krN-QIGYvg_RsByGec/\
  aggregate_shares/lc7aUeGpdSNosNlh-UZhKA
Host: example.com
Authorization: Bearer auth-token

HTTP/1.1 200
Retry-After: 300

GET /helper/tasks/8BY0RzZMzxvA46_8ymhzycOB9krN-QIGYvg_RsByGec/\
  aggregate_shares/lc7aUeGpdSNosNlh-UZhKA
Host: example.com
Authorization: Bearer auth-token

HTTP/1.1 200
Content-Type: application/ppm-dap;message=aggregate-share

encoded(struct {
  encrypted_aggregate_share = struct { ... } HpkeCiphertext,
} AggregateShare)
```

4.6.4. Aggregate Share Deletion

The Leader can send a DELETE request to the aggregate share, which indicates to the Helper that it can abandon the aggregate share and discard state related to it.

Aggregators MUST NOT delete information needed for replay or double collection checks (Section 4.5.3.3).

4.6.4.1. Example

```
DELETE /helper/tasks/8BY0RzZMzxvA46_8ymhzycOB9krN-QIGYvg_RsByGec/\
  aggregate_shares/lc7aUeGpdSNosNlh-UZhKA
Host: example.com
Authorization: Bearer auth-token

HTTP/1.1 200
```

4.6.5. 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 4.6.6. Once the Collector successfully decrypts all aggregate shares, it unshards the aggregate shares into an aggregate result using the VDAF's unshard algorithm.

Let leader_agg_share denote the Leader's aggregate share, helper_agg_share denote the Helper's aggregate share, report_count denote the report count sent by the Leader, and agg_param denote 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)
```

4.6.6. Aggregate Share Encryption

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

```
(enc, payload) = SealBase(
    pk,
    "dap-17 aggregate share" || server_role || 0x00,
    agg_share_aad,
    agg_share)
```

* pk is the Collector's HPKE public 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)

* agg_share_aad is an AggregateShareAad (defined below).

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-17 aggregate share" || server_role || 0x00,
    agg_share_aad,
    enc_share.payload)
```

* 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)

* 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 determined by the batch mode:

  - For time-interval (Section 5.1), the batch selector is the batch interval specified in the query.

  - For leader-selected (Section 5.2), the batch selector is the batch ID sent in the response.

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

5. Batch Modes

This section defines an initial set of batch modes for DAP. New batch modes may be defined by future documents following the guidelines in Section 10.

In protocol messages, batch modes are identified with a BatchMode value:

```
enum {
  reserved(0),
  time_interval(1),
  leader_selected(2),
  (255)
} BatchMode;
```

Each batch mode specifies the following:

1. The value of the config field of Query, PartialBatchSelector, and BatchSelector

2. Batch buckets (Section 4.5.3.3): how reports are assigned to batch buckets; how each bucket is identified; and how batch buckets are mapped to batches

5.1. Time Interval

The time-interval batch mode is designed to support applications in which reports are grouped by an interval of time. The Collector specifies a "batch interval" into which report timestamps must fall.

The Collector can issue queries whose batch intervals are continuous, monotonically increasing, and have the same duration. For example, the following sequence of batch intervals satisfies these conditions:

```
[
  struct {
    start = 1659544,
    duration = 1,
  } Interval,
  struct {
    start = 1659545,
    duration = 1,
  } Interval,
  struct {
    start = 1659546,
    duration = 1,
  } Interval,
  struct {
    start = 1659547,
    duration = 1,
  } Interval,
]
```

However, this is not a requirement: the Collector may decide to issue queries out-of-order. In addition, the Collector may need to vary the duration to adjust to changing report upload rates.

5.1.1. Query Configuration

The payload of Query.config is

```
struct {
  Interval batch_interval;
} TimeIntervalQueryConfig;
```

where batch_interval is the batch interval requested by the Collector. The interval MUST be well-formed as specified in Section 4.1.1. Otherwise, the query does not specify a set of valid batch buckets.

5.1.2. Partial Batch Selector Configuration

The payload of PartialBatchSelector.config is empty.

5.1.3. Batch Selector Configuration

The payload of BatchSelector.config is

```
struct {
  Interval batch_interval;
} TimeIntervalBatchSelectorConfig;
```

where batch_interval is the batch interval requested by the Collector.

5.1.4. Batch Buckets

Each batch bucket is identified by an Interval whose duration is equal to the task's time_precision. The identifier associated with a given report is the unique such interval containing the timestamp of the report. For example, if the task's time_precision is 1000 seconds and the report was generated at 1729629081 seconds after the start of the UNIX epoch, the relevant batch bucket identifier is

```
struct {
  start = 1729629,
  duration = 1,
} Interval
```

The Query received by the Leader or BatchSelector received by the Helper determines a valid set of batch bucket identifiers if the batch interval's duration is greater than or equal to the task's time_precision.

A batch consists of a sequence of contiguous batch buckets. That is, the set of batch bucket identifiers for the batch interval is

```
[
  struct {
    start = batch_interval.start,
    duration = 1,
  } Interval,
  struct {
    start = batch_interval.start + 1,
    duration = 1,
  } Interval,
  ...
  struct {
    start = batch_interval.start + batch_interval.duration - 1,
    duration = 1,
  } Interval,
]
```

5.2. Leader-selected Batch Mode

The leader-selected batch mode is used when it is acceptable for the Leader to arbitrarily batch reports. Each batch is identified by an opaque "batch ID" chosen by the Leader, which MUST be unique in the scope of the task.

```
opaque BatchID[32];
```

The Collector will not know the set of batch IDs available for collection. To get the aggregate of a batch, the Collector issues a query for the next available batch. The Leader selects a recent batch to aggregate which MUST NOT yet have been associated with a collection job.

The Aggregators can output batches of any size that is larger than or equal to the task's minimum batch size. The target batch size, if any, is implementation-specific, and may be equal to or greater than the minimum batch size. Deciding how soon batches should be output is also implementation-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.

5.2.1. Query Configuration

They payload of Query.config is empty. The request merely indicates the Collector would like the next batch selected by the Leader.

5.2.2. Partial Batch Selector Configuration

The payload of PartialBatchSelector.config is:

```
struct {
  BatchID batch_id;
} LeaderSelectedPartialBatchSelectorConfig;
```

where batch_id is the batch ID selected by the Leader.

5.2.3. Batch Selector Configuration

The payload of BatchSelector.config is:

```
struct {
  BatchID batch_id;
} LeaderSelectedBatchSelectorConfig;
```

where batch_id is the batch ID selected by the Leader.

5.2.4. Batch Buckets

Each batch consists of a single bucket and is identified by the batch ID. A report is assigned to the batch indicated by the PartialBatchSelector during aggregation.

6. 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.

6.1. 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.

6.1.1. 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.

6.1.2. 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 interaction.

* 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 4.5.2.4.

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

* Support the upload interaction and store reports; and

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

* Implement and store state for the form of inter- and intra-batch replay mitigation in Figure 7. This requires storing the report IDs of all reports processed for a given task. Implementations may find it helpful to track additional information, like the timestamp, so that the storage used for anti-replay can be sharded efficiently.

6.1.3. 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.

6.2. VDAFs and Compute Requirements

The choice of VDAF can impact the computation and storage required for a DAP task:

* The runtime of VDAF sharding and verification is related to the "size" of the underlying measurements. For example, the Prio3SumVec VDAF defined in Section 7 of [VDAF] requires each measurement to be a vector of the same length, which all parties need to agree on prior to VDAF execution. The computation required for such tasks increases linearly as a function of the chosen length, as each vector element must be processed in turn.

* The runtime of VDAF verification is related to the size of the aggregation parameter. For example for Poplar1 defined in Section 8 of [VDAF], verification takes as input a sequence of so-called "candidate prefixes", and the amount of computation is linear in the number of prefixes.

* The storage requirements for aggregate shares vary depending on the size of the measurements and/or the aggregation parameter.

To account for these factors, care must be taken that a DAP deployment can handle VDAF execution of all possible configurations for any tasks which the deployment may be configured for. Otherwise, an attacker may deny service by uploading many expensive reports to a suitably-configured VDAF.

The varying cost of VDAF computation means that Aggregators should negotiate reasonable limits for each VDAF configuration, out of band with the protocol. For example, Aggregators may agree on a maximum size for an aggregation job or on a maximum rate of incoming 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.

6.3. 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 VDAF. 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.

6.4. 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.

6.4.1. Reducing Storage Requirements

In general, the Aggregators are required to keep state for tasks and all valid reports for as long as collection requests can be made for them. However, it is not necessary to store the complete reports. Each Aggregator only needs to store an aggregate share for each possible batch bucket i.e., the batch interval for time-interval or batch ID for leader-selected, along with a flag indicating whether the aggregate share has been collected. This is due to the requirement for queries to respect bucket boundaries. See Section 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 4.5.2.4. 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 is not after this task's task_interval. Aggregators MAY delete the task and all data pertaining to this task after the task_interval. Implementors SHOULD provide for some leeway so the Collector can collect the batch after some delay.

6.4.2. Distributed Systems and Synchronization Concerns

Various parts of a DAP implementation will need to synchronize in order to ensure correctness during concurrent operation. This section describes the relevant concerns and makes suggestions as to potential implementation tradeoffs.

* The upload interaction requires the Leader to discard uploaded reports with a duplicated ID, including concurrently-uploaded reports. This might be implemented by synchronization or via an eventually-consistent process. If the Leader wishes to alert the Client with a reportRejected error, synchronization will be necessary to ensure all but one concurrent request receive the error.

* The Leader is responsible for generating aggregation jobs, and will generally want to place each report in exactly one aggregation job. (The only event in which a Leader can place a report in multiple aggregation jobs is if the Helper rejects the report with report_too_early, in which case the Leader can place

the report into a later aggregation job.) This may require synchronization between different components of the system which are generating aggregation jobs. Note that placing a report into more than one aggregation job will result in a loss of throughput, rather than a loss of correctness, privacy, or verifiability, so it is acceptable for implementations to use an eventually-consistent scheme which may rarely place a report into multiple aggregation jobs.

* Aggregation is implemented as a sequence of aggregation steps by both the Leader and the Helper. The Leader must ensure that each aggregation job is only processed once concurrently, which may require synchronization between the components responsible for performing aggregation. The Helper must ensure that concurrent requests against the same aggregation job are handled appropriately, which requires synchronization between the components handling aggregation requests.

* Aggregation requires checking and updating used-report storage as part of implementing replay protection. This must be done while processing the aggregation job, though which steps the checks are performed at is up to the implementation. The checks and storage require synchronization, so that if two aggregation jobs containing the same report are processed, at most one instance of the report will be aggregated. However, the interaction with the used-report storage does not necessarily have to be synchronized with the processing and storage for the remainder of the aggregation process. For example, used-report storage could be implemented in a separate datastore than is used for the remainder of data storage, without any transactionality between updates to the two datastores.

* The aggregation and collection interactions require synchronization to avoid modifying the aggregate of a batch after it has already been collected. Any reports being aggregated which pertain to a batch which has already been collected must fail with a batch_collected error; correctly determining this requires synchronizing aggregation with the completion of collection jobs (for the Leader) or aggregate share requests (for the Helper). Also, the Leader must complete all outstanding aggregation jobs for a batch before requesting aggregate shares from the Helper, again requiring synchronization between the Leader’s collection and aggregation interactions. Further, the Helper must determine the aggregated report count and checksum of aggregated report IDs before responding to an aggregate share request, requiring synchronization between the Helper’s collection and aggregation interactions.

6.4.3. Streaming Messages

Most messages in the protocol contain fixed-length or length-prefixed fields such that they can be parsed independently of context. The exceptions are the UploadReq, UploadErrors (Section 4.4.2), AggregationJobInitReq, AggregationJobContinueReq, and AggregationJobResp (Section 4.5) messages, all of which contain vectors whose length is determined by the length of the enclosing HTTP message.

This allows implementations to begin transmitting these messages before knowing how long the message will ultimately be. This is useful if implementations wish to avoid buffering exceptionally large messages in memory.

7. 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)

8. Security Considerations

DAP aims to achieve the privacy and verifiability security goals defined in Section 9 of [VDAF]. That is, an active attacker that controls a subset of the Clients, one of the Aggregators, and the Collector learns nothing about the honest Clients' measurements beyond their aggregate result. At the same time, an attacker that controls a subset of Clients cannot force the Collector to compute anything but the aggregate result over the honest Clients' measurements.

Since DAP requires HTTPS (Section 3), the attacker cannot tamper with messages delivered by honest parties or forge messages from honest, authenticated parties; but it can drop messages or forge messages from unauthenticated parties. Thus there are some threats that DAP does not defend against and which are considered outside of its threat model. These and others are enumerated below, along with potential mitigations.

Attacks on verifiability:

1. Aggregators can change the result by an arbitrary amount by emitting incorrect aggregate shares, by omitting reports from the aggregation process, or by manipulating the VDAF verification process for a single report. Like the underlying VDAF, DAP only ensures correct computation of the aggregate result if both Aggregators honestly execute 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 8.1 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 8.4), 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. Some VDAFs could leak information to either Aggregator or the Collector beyond what the protocol intended to learn. It may be possible to mitigate such leakages using differential privacy (Section 8.5).

8.1. Sybil Attacks

Several attacks on the security of the VDAF (Section 9 of [VDAF]) 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 8.3, 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 8.4.

3. Some mechanisms for differential privacy (Section 8.5) can help mitigate Sybil attacks against privacy to some extent.

8.2. Batch-selection Attacks

Depending on the batch mode, the privacy of an individual Client may be infringed upon by selection of the batch. For example, in the leader-selected batch mode, the Leader is free to select the reports that compose a given batch almost arbitrarily; a malicious Leader might choose a batch composed of reports arriving from a single client. The aggregate derived from this batch might then reveal information about that Client.

The mitigations for this attack are similar to those used for Sybil attacks (Section 8.1):

1. Implementing Client authentication, as described in Section 8.3, and having each aggregator verify that each batch contains reports from a suitable number of distinct clients.

2. Disassociating each report from the Client which generated it, via the use of anonymizing proxies (Section 8.4) or similar techniques.

3. Differential privacy (Section 8.5) can help mitigate the impact of this attack.

4. Deployment-specific mitigations may also be possible: for example, if every Client is sending reports at a given rate, it may be possible for aggregators to bound the accepted age of reports such that the number of aggregatable reports from a given Client is small enough to effectively mitigate this attack.

8.3. 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 8.4) 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 ([RFC9576]) in a report 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.

8.4. Anonymizing Proxies

Client reports may be transmitted alongside auxiliary information such as source IP, HTTP user agent, or Client authentication information (in deployments which use it, see Section 8.3). This metadata can be used by Aggregators to identify participating Clients or permit some attacks on verifiability. 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.

The report itself may contain deanonymizing information that cannot be removed by a proxy:

* The report timestamp indicates when a report was generated and may help an attacker to deduce which Client generated it. Truncating this timestamp as described in Section 4.1.1 can help.

* The public extensions may help the attacker to profile the Client’s configuration.

8.5. 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.

8.6. 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).

8.6.1. Predictable or Enumerable Task IDs

This specification imposes no requirements on task IDs except that they be globally unique. One way to achieve this is to use random task IDs, but deployments can also use schemes like [I-D.draft-ietf-ppm-dap-taskprov-03] where task IDs are deterministically generated from some set of task parameters.

In such settings, deployments should consider whether an Aggregator acknowledging the existence of a task (by accepting report uploads or aggregation jobs, for example) could unintentionally leak information such as a label describing the task, the identities of participating Aggregators or the fact that some measurement is being taken at all.

Such enumeration attacks can be mitigated by incorporating unpredictable values into the task ID derivation. They do not, however, affect the core security goals of VDAFs (Section 9 of [VDAF]).

8.6.2. 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 verification. 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.

8.6.3. 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 collection, 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 8.5) parameters in use, if any.

8.6.4. Relaxing Report Processing Rules

DAP Aggregators enforce several rules for report processing related to the privacy of individual measurements:

1. Each report may be aggregated at most once (Section 4.5.3.3)

2. A batch bucket may be collected at most once (reports pertaining to collected buckets are rejected; see Section 4.5.3.3)

3. A batch may only be collected if the number of reports aggregated exceeds the minimum batch size (Section 4.6)

It may be desirable to relax these rules in some applications. It may also be safe to do so when DAP is combined with other privacy enhancements such as differential privacy. When applications wish to relax any of one of these requirements, they:

1. MUST adhere to the VDAF’s requirements for aggregating a report more than once. See Section 5.3 of [VDAF] for details.

2. SHOULD define a mechanism by which each party explicitly opts into the change in report processing rules, e.g., via a report extension (Section 4.4.3). This helps prevent an implementation from relaxing the rules by mistake.

8.6.5. 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 report 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.

8.7. 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.

9. IANA Considerations

This document requests registry of a new media type (Section 9.1), creation of new codepoint registries (Section 9.2), and registration of an IETF URN sub-namespace (Section 9.3).

(RFC EDITOR: In the remainder of this section, replace "RFC XXXX" with the RFC number assigned to this document.)

9.1. Protocol Message Media Type

This specification defines a new media type used for all protocol messages: application/ppm-dap. Specific message types are distinguished using a message parameter.

This specification defines the following protocol messages, along with their corresponding message value:

* HpkeConfigList Section 4.4.1: "hpke-config-list"

* UploadRequest Section 4.4.2: "upload-req"

* UploadErrors Section 4.4.2: "upload-errors"

* AggregationJobInitReq Section 4.5.2.1: "aggregation-job-init-req"

* AggregationJobResp Section 4.5.2.2: "aggregation-job-resp"

* AggregationJobContinueReq Section 4.5.3.1: "aggregation-job-continue-req"

* AggregateShareReq Section 4.6.3: "aggregate-share-req"

* AggregateShare Section 4.6.3: "aggregate-share"

* CollectionJobReq Section 4.6.1: "collection-job-req"

* CollectionJobResp Section 4.6.1: "collection-job-resp"

For example, a request whose body consists of an AggregationJobInitReq could have the header Content-Type: application/ppm-dap;message=aggregation-job-init-req.

Protocol message format evolution is supported through the definition of new formats that are identified by message values. The messages above are specific to this specification. When a new major enhancement is proposed that results in newer IETF specification for DAP, a new media type will be defined. In other words, newer versions of DAP will not be backward compatible with this version of DAP.

(RFC EDITOR: Remove this paragraph.) HTTP requests with DAP media types MAY express an optional parameter 'version', following Section 8.3 of [RFC9110]. Value of this parameter indicates current draft version of the protocol the component is using. This MAY be used as a hint by the receiver of the request to do compatibility checks between client and server. For example, A report submission to leader from a client that supports draft-ietf-ppm-dap-09 could have the header Content-Type: application/ppm-dap;message=upload-req;version=09.

The "Media Types" registry at https://www.iana.org/assignments/media-types will be (RFC EDITOR: replace "will be" with "has been") updated to include the application/ppm-dap media type.

Type name: application

Subtype name: ppm-dap

Required parameters: message

Optional parameters: None

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

Security considerations: see Section 8 of the published specification

Interoperability considerations: N/A

Published specification: RFC XXXX

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: PPM WG mailing list (ppm@ietf.org)

Intended usage: COMMON

Restrictions on usage: N/A

Author: see Authors' Addresses section of the published specification

Change controller: IETF

9.2. DAP Type Registries

This document also requests creation of a new "Distributed Aggregation Protocol (DAP)" page. This page will contain several new registries, described in the following sections. All registries are administered under the Specification Required policy [RFC8126].

9.2.1. Batch Modes Registry

A new registry will be (RFC EDITOR: change "will be" to "has been") created called "DAP Batch Mode Identifiers" (Section 5). This registry should contain the following columns:

Value: The one-byte identifier for the batch mode

Name: The name of the batch mode

Reference: Where the batch mode is defined

The initial contents of this registry listed in Table 2.

Value	Name	Reference
0x00	reserved	Section 5 of RFC XXXX
0x01	time_interval	Section 5.1 of RFC XXXX
0x02	leader_selected	Section 5.2 of RFC XXXX

Table 2: Initial contents of the DAP Batch Mode Identifiers registry.

9.2.2. Report Extension Registry

A new registry will be (RFC EDITOR: change "will be" to "has been") created called "DAP Report Extension Identifiers" for extensions to the report structure (Section 4.4.3). This registry should contain the following columns:

Value: The two-byte identifier for the upload extension

Name: The name of the upload extension

Reference: Where the upload extension is defined

The initial contents of this registry are listed in Table 3.

Value	Name	Reference
0x0000	reserved	RFC XXXX

Table 3: Initial contents of the DAP Report Extension Identifiers registry.

9.2.3. Report Error Registry

A new registry will be (RFC EDITOR: change "will be" to "has been") created called "DAP Report Error Identifiers" for reasons for rejecting reports during the aggregation interaction (Section 4.5.2.2).

Value: The one-byte identifier of the report error

Name: The name of the report error

Reference: Where the report error is defined

The initial contents of this registry are listed below in Table 4.

Value	Name	Reference
0x00	reserved	Section 4.1 of RFX XXXX
0x01	batch_collected	Section 4.1 of RFX XXXX
0x02	report_replayed	Section 4.1 of RFX XXXX
0x03	report_dropped	Section 4.1 of RFX XXXX
0x04	hpke_unknown_config_id	Section 4.1 of RFX XXXX
0x05	hpke_decrypt_error	Section 4.1 of RFX XXXX
0x06	vdaf_verify_error	Section 4.1 of RFX XXXX
0x07	task_expired	Section 4.1 of RFX XXXX
0x08	invalid_message	Section 4.1 of RFX XXXX
0x09	report_too_early	Section 4.1 of RFX XXXX
0x0A	task_not_started	Section 4.1 of RFX XXXX
0x0B	outdated_config	Section 4.1 of RFX XXXX

Table 4: Initial contents of the DAP Report Error Identifiers registry.

9.2.4. Guidance for Designated Experts

When reviewing requests to extend a DAP registry, experts should ensure that a document exists that defines the newly registered items and review the document to ensure it meets the criteria for extending DAP specified in Section 10.

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

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

Registry name: dap

Specification: RFC XXXX

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

Index value: No transformation needed.

The initial contents of this namespace are the types and descriptions in Table 1, with the Reference field set to RFC XXXX.

10. Extending this Document

The behavior of DAP may be extended or modified by future documents defining one or more of the following:

1. a new batch mode (Section 5)

2. a new report extension (Section 4.4.3)

3. a new report error (Section 4.5.2.2)

4. a new entry in the URN sub-namespace for DAP (Table 1)

Each of these requires registration of a codepoint or other value; see Section 9. No other considerations are required except in the following cases:

* When a document defines a new batch mode, it MUST include a section titled "DAP Batch Mode Considerations" specifying the following:

  - The value of the config field of Query, PartialBatchSelector, and BatchSelector

  - Batch buckets (Section 4.5.3.3): how reports are assigned to batch buckets; how each bucket is identified; and how batch buckets are mapped to batches.

  See Section 5 for examples.

* When a document defines a new report extension, it SHOULD include in its "Security Considerations" section some discussion of how the extension impacts the security of DAP with respect to the threat model in Section 8.

11. References

11.1. 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>.

[POSIX] "IEEE Standard for Information Technology--Portable Operating System Interface (POSIX(TM)) Base Specifications, Issue 7", IEEE, DOI 10.1109/ieeestd.2018.8277153, ISBN ["9781504445429"], January 2018, <https://doi.org/10.1109/ieeestd.2018.8277153>.

[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>.

[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>.

[RFC8126] Cotton, M., Leiba, B., and T. Narten, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 8126, DOI 10.17487/RFC8126, June 2017, <https://www.rfc-editor.org/rfc/rfc8126>.

[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>.

[RFC8792] Watsen, K., Auerswald, E., Farrel, A., and Q. Wu, "Handling Long Lines in Content of Internet-Drafts and RFCs", RFC 8792, DOI 10.17487/RFC8792, June 2020, <https://www.rfc-editor.org/rfc/rfc8792>.

[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] "Secure hash standard", National Institute of Standards and Technology (U.S.), DOI 10.6028/nist.fips.180-4, 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-18, 30 January 2026, <https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-vdaf-18>.

11.2. Informative References

[Dou02] Douceur, J. R., "The Sybil Attack", International Workshop on Peer-to-Peer Systems (IPTPS), 2002, <https://doi.org/10.1007/3-540-45748-8_24>.

[I-D.draft-ietf-ppm-dap-taskprov-03] Wang, S. and C. Patton, "Task Binding and In-Band Provisioning for DAP", Work in Progress, Internet-Draft, draft-ietf-ppm-dap-taskprov-03, 5 September 2025, <https://datatracker.ietf.org/doc/html/draft-ietf-ppm-dap-taskprov-03>.

[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>.

[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>.

[RFC9421] Backman, A., Ed., Richer, J., Ed., and M. Sporny, "HTTP Message Signatures", RFC 9421, DOI 10.17487/RFC9421, February 2024, <https://www.rfc-editor.org/rfc/rfc9421>.

[RFC9576] Davidson, A., Iyengar, J., and C. A. Wood, "The Privacy Pass Architecture", RFC 9576, DOI 10.17487/RFC9576, June 2024, <https://www.rfc-editor.org/rfc/rfc9576>.

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

Appendix A. HTTP Resources Reference

This appendix enumerates all the HTTP resources this document describes, the HTTP methods that they support and media-types that may occur in requests and responses. It is organized by the protocol role that serves the resource.

This section is intended to act as a reference and checklist for implementers. The HTTP methods and media-types described in this appendix are not authoritative. See the section that each resource refers to for detailed specifications.

A.1. Aggregator

A.1.1. HPKE Configurations

Aggregator HPKE configurations to which Clients will encrypt report shares.

Resource URL: {aggregator}/hpke_config

HTTP methods: GET

Request media-type message values: N/A

Response media-type message values: hpke-config-list

Reference: Section 4.4.1

A.2. Leader

A.2.1. Reports

Reports being uploaded to the Leader by the Client.

Resource URL: {leader}/tasks/{task-id}/reports

HTTP methods: POST

Request media-type message values: upload-req

Response media-type message values: upload-errors

Reference: Section 4.4.2

A.2.2. Collection Jobs

A Collector’s request to collect reports identified by some query.

Resource URL: {leader}/tasks/{task-id}/collection_jobs/{collection-job-id}

HTTP methods: PUT, GET

Request media-type message values: collection-job-req

Response media-type message values: collection-job-resp

Reference: Section 4.6

A.3. Helper

A.3.1. Aggregation Jobs

An aggregation job created by the Leader.

Resource URL: /tasks/{task-id}/aggregation_jobs/{aggregation-job-id}

HTTP methods: PUT, POST, GET

Request media-type message values: aggregation-job-init-req, aggregation-job-continue-req

Response media-type message values: aggregation-job-resp

Reference: Section 4.5

A.3.2. Aggregate Shares

The Helper's share of an aggregation over the reports identified by the Collector's query.

Resource URL: {helper}/tasks/{task-id}/aggregate_shares/{aggregate-share-id}

HTTP methods: PUT, GET

Request media-type message values: aggregate-share-req

Response media-type message values: aggregate-share

Reference: Section 4.6.3

Contributors

Josh Aas
ISRG
Email: josh@abetterinternet.org

Junye Chen
Apple
Email: junyec@apple.com

David Cook
ISRG
Email: dcook@divviup.org

Suman Ganta
Apple
Email: sganta2@apple.com

Ameer Ghani
ISRG
Email: inahga@divviup.org

Kristine Guo
Apple
Email: kristine_guo@apple.com

Charlie Harrison
Google
Email: csharrison@chromium.org

J.C. Jones
ISRG
Email: ietf@insufficient.coffee

Alex Koshelev
Meta
Email: koshelev@meta.com

Peter Saint-Andre
Email: stpeter@gmail.com

Shivan Sahib
Brave
Email: shivankaulsahib@gmail.com

Phillipp Schoppmann
Google
Email: schoppmann@google.com

Martin Thomson
Mozilla
Email: mt@mozilla.com

Shan Wang
Apple
Email: shan_wang@apple.com

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
Independent
Email: ekr@rtfm.com

Christopher A. Wood
Cloudflare
Email: caw@heapingbits.net