Workgroup: Network Working Group
Internet-Draft: draft-ietf-ppm-dap-03
Published: 10 December 2022
Intended Status: Standards Track
Expires: 13 June 2023
Authors: T. Geoghegan ISRG
C. Patton Cloudflare
E. Rescorla Mozilla
C. A. Wood Cloudflare

Distributed Aggregation Protocol for Privacy Preserving Measurement

Abstract

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

About This Document

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

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

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

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

Status of This Memo

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

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

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

This Internet-Draft will expire on 13 June 2023.

Copyright Notice

Table of Contents

1. Introduction

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

1.1. Change Log

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

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.

The following terms are used:

Aggregate result: The output of the aggregation function over a given set of reports.

Aggregate share: A share of the aggregate result emitted by an aggregator. Aggregate shares are reassembled by the collector into the final output.

Aggregation function: The function computed over the users' inputs.

Aggregator: An endpoint that runs the input-validation protocol and accumulates input shares.

Batch: A set of reports that are aggregated into an output.

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

Batch interval: A parameter of the collect or aggregate-share request that specifies the time range of the reports in the batch.

Client: The endpoint from which a user sends data to be aggregated, e.g., a web browser.

Collector: The endpoint that receives the output of the aggregation function.

Helper: Executes the protocol as instructed by the leader.

Input: The measurement (or measurements) emitted by a client, before any encryption or secret sharing scheme is applied.

Input share: An aggregator's share of the output of the VDAF [VDAF] sharding algorithm. This algorithm is run by each client in order to cryptographically protect its measurement.

Leader: A distinguished aggregator that coordinates input validation and data collection.

Measurement: A single value (e.g., a count) being reported by a client. Multiple measurements may be grouped into a single protocol input.

Minimum batch duration: The minimum batch duration permitted for a DAP task, i.e., the minimum time difference between the oldest and newest report in a batch.

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

Output share: An aggregator's share of the output of the VDAF [VDAF] preparation step. Many output shares are combined into an aggregate share via the VDAF aggregation algorithm.

Proof: A value generated by the client and used by the aggregators to verify the client's input.

Report: Uploaded to the leader from the client. A report contains the secret-shared and encrypted input and proof.

Server: An aggregator.

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

2. Overview

The protocol is executed by a large set of clients and a small set of servers. Servers are referred to as *aggregators*. Each client's input to the protocol is a set of measurements (e.g., counts of some user behavior). Given the input set of measurements x_1, ..., x_n held by n users, the goal of a protocol for privacy preserving measurement is to compute y = F(p, x_1, ..., x_n) for some function F while revealing nothing else about the measurements.

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

* Prio3, which allows for aggregate statistics such as sum, mean, histograms, etc. This class of VDAFs is based on Prio [CGB17] and includes improvements described in [BBCGGI19].

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

This protocol is designed to work with schemes that use secret sharing. Rather than sending its input in the clear, each client shards its measurements into a sequence of *input shares* and sends an input share to each of the aggregators. This provides two important properties:

* It is impossible to deduce the measurement without knowing *all* of the shares.

*It allows the aggregators to compute the final output by first aggregating up their measurements shares locally, then combining the results to obtain the final output.

2.1. System Architecture

The overall system architecture is shown in Figure 1.

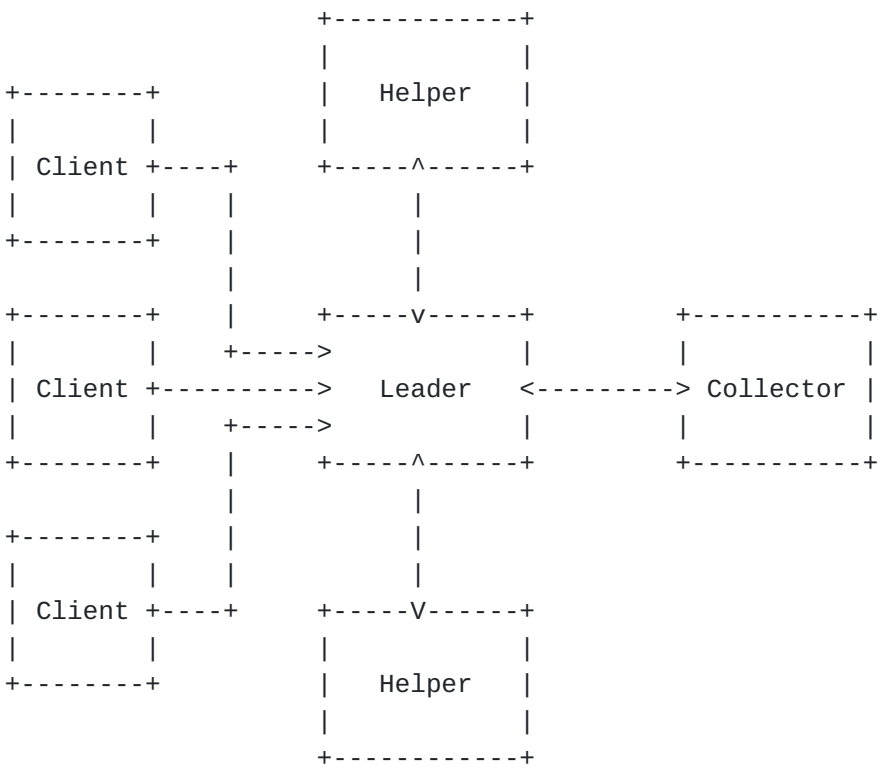

Figure 1: System Architecture

[[OPEN ISSUE: This shows two helpers, but the document only allows one for now. https://github.com/ietf-wg-ppm/draft-ietf-ppm-dap/issues/117]]

The main participants in the protocol are as follows:

Collector: The entity which wants to take the measurement and ultimately receives the results. Any given measurement will have a single collector.

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

Aggregator: An endpoint which receives report shares. Each aggregator works with the other aggregators to compute the final aggregate. This protocol defines two types of aggregators:

Leaders and Helpers. For each measurement, there is a single leader and helper.

Leader: The leader is responsible for coordinating the protocol. It receives the encrypted shares, distributes them to the helpers, and orchestrates the process of computing the final measurement as requested by the collector.

Helper: Helpers are responsible for executing the protocol as instructed by the leader. The protocol is designed so that helpers can be relatively lightweight, with most of the state held at the leader.

The basic unit of DAP is the "task" which represents a single measurement (though potentially taken over multiple time windows). The definition of a task includes the following parameters:

- The type of each measurement.
- The aggregation function to compute (e.g., sum, mean, etc.).
- The set of aggregators and necessary cryptographic keying material to use.
- The VDAF to execute, which to some extent is dictated by the previous choices.
- The minimum "batch size" of reports which can be aggregated.
- The rate at which measurements can be taken, i.e., the "minimum batch window".

These parameters are distributed out of band to the clients and to the aggregators. They are distributed by the collecting entity in some authenticated form. Each task is identified by a unique 32-byte ID which is used to refer to it in protocol messages.

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

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

2.2. Validating Inputs

An essential task of any data collection pipeline is ensuring that the data being aggregated is "valid". In DAP, input validation is complicated by the fact that none of the entities other than the client ever sees the values for individual clients.

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

To facilitate this computation, the input shares generated by the client include information used by the aggregators during aggregation in order to validate their corresponding output shares. For example, Prio3 includes a distributed zero-knowledge proof of the input's validity [BBCGGI19] which the aggregators can jointly verify and reject the report if it cannot be verified. However, they do not learn anything about the individual report other than that it is valid.

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

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

3. Message Transport

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

3.1. HTTPS Request Authentication

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

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

In other cases, DAP requires HTTPS client authentication. Any authentication scheme that is composable with HTTP is allowed. For example, [OAuth2] credentials are presented in an Authorization HTTP header, which can be added to any DAP protocol message, or TLS client certificates are another viable solution. This allows organizations deploying DAP to use existing well-known HTTP authentication mechanisms that they already support. Discovering what authentication mechanisms are supported by a DAP participant is outside of this document's scope.

3.2. Errors

Errors can be reported in DAP both at the HTTP layer and within challenge objects as defined in Section 8. DAP servers can return responses with an HTTP error response code (4XX or 5XX). For example, if the client submits a request using a method not allowed in this document, then the server **MAY** return HTTP status code 405 Method Not Allowed.

When the server responds with an error status, it **SHOULD** provide additional information using a problem document [RFC7807]. To facilitate automatic response to errors, this document defines the following standard tokens for use in the "type" field (within the DAP URN namespace "urn:ietf:params:ppm:dap:error:"):

Type	Description
unrecognizedMessage	The message type for a response was incorrect or the payload was malformed.
unrecognizedTask	An endpoint received a message with an unknown task ID.
unrecognizedAggregationJob	An endpoint received a message with an unknown aggregation job ID.
outdatedConfig	The message was generated using an outdated configuration.
reportRejected	Report could not be processed for an unspecified reason.
reportTooEarly	Report could not be processed because its timestamp is too far in the future.
batchInvalid	A collect or aggregate-share request was made with invalid batch parameters.
invalidBatchSize	There are an invalid number of reports in the batch.
batchQueriedTooManyTimes	

Type	Description
	The maximum number of batch queries has been exceeded for one or more reports included in the batch.
batchMismatch	Aggregators disagree on the report shares that were aggregated in a batch.
unauthorizedRequest	Authentication of an HTTP request failed (see Section 3.1).
missingTaskID	HPKE configuration was requested without specifying a task ID.
queryMismatch	Query type indicated by a message does not match the task's query type.

Table 1

This list is not exhaustive. The server **MAY** return errors set to a URI other than those defined above. Servers **MUST NOT** use the DAP URN namespace for errors not listed in the appropriate IANA registry (see Section 8.4). 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 'unrecognizedMessage'" refers to an error document with "type" value "urn:ietf:params:ppm:dap:error:unrecognizedMessage".

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

4. Protocol Definition

DAP has three major interactions which need to be defined:

- Uploading reports from the client to the aggregators, specified in Section 4.3
- Computing the results of a given measurement, specified in Section 4.4
- Collecting aggregated results, specified in Section 4.5

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

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

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

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

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

/* An ID used to uniquely identify a report in the context of a DAP task
uint8 ReportID[16];

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

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

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

/* Represent a zero byte empty data type. */
struct {} Empty;
```

4.1. Queries

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

This document defines the following query types:

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

The time_interval query type is described in Section 4.1.1; the fixed_size query type is described in Section 4.1.2. Future specifications can introduce new query types as needed (see Section 8.2). A query includes parameters used by the Aggregators to select a batch of reports specific to the given query type. A query is defined as follows:

```
opaque BatchID[32];

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

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

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

The parameters pertaining to each query type are described in one of the subsections below. The query is issued in-band as part of the collect sub-protocol (Section 4.5). Its content is determined by the "query type", which in turn is encoded by the "query configuration" configured out-of-band. All query types have the following configuration parameters in common:

* min_batch_size - The smallest number of reports the batch is allowed to include. In a sense, this parameter controls the degree of privacy that will be obtained: The larger the minimum batch size, the higher degree of privacy. However, this

ultimately depends on the application and the nature of the reports and aggregation function.

* time_precision - Clients use this value to truncate their report timestamps; see Section 4.3. Additional semantics may apply, depending on the query type. (See Section 4.5.6 for details.)

The parameters pertaining to specific query types are described in the relevant subsection below.

4.1.1. Time-interval Queries

The first query type, time_interval, is designed to support applications in which reports are collected over a long period of time. The Collector specifies a "batch interval" that determines the time range for reports included in the batch. For each report in the batch, the time at which that report was generated (see Section 4.3) must fall within the batch interval specified by the Collector.

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

4.1.2. Fixed-size Queries

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

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

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

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

Implementation note: The goal for the Aggregators is to aggregate precisely min_batch_size reports per batch. Doing so, however, may be challenging for Leader deployments in which multiple, independent nodes running the aggregate sub-protocol (see Section 4.4) need to be coordinated. The maximum batch size is intended to allow room for error. Typically the difference between the minimum and maximum batch size will be a small fraction of the target batch size for each batch.

[OPEN ISSUE: It may be feasible to require a fixed batch size, i.e., min_batch_size == max_batch_size. We should know better once we've had some implementation/deployment experience.]

4.2. Task Configuration

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

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

A TaskID is a globally unique sequence of bytes. It is **RECOMMENDED** that this be set to a random string output by a cryptographically secure pseudorandom number generator. Each task has the following parameters associated with it:

* aggregator_endpoints: A list of URLs relative to which an aggregator's API endpoints can be found. Each endpoint's list **MUST** be in the same order. The leader's endpoint **MUST** be the first in the list. The order of the encrypted_input_shares in a Report (see Section 4.3) **MUST** be the same as the order in which aggregators appear in this list.

* The query configuration for this task (see Section 4.1). This determines the query type for batch selection and the properties that all batches for this task must have.

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

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

* A unique identifier for the VDAF instance used for the task, including the type of measurement associated with the task.

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

* collector_config: The [HPKE] configuration of the collector (described in Section 4.3.1); see Section 6 for information about the HPKE configuration algorithms.

* vdaf_verify_key: The VDAF verification key shared by the aggregators. This key is used in the aggregation sub-protocol (Section 4.4). [OPEN ISSUE: The manner in which this key is distributed may be relevant to the VDAF's security. See issue#161.]

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

4.3. Uploading Reports

Clients periodically upload reports to the leader, which then distributes the individual shares to each helper.

4.3.1. HPKE Configuration Request

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

Clients retrieve the HPKE configuration from each aggregator by sending an HTTP GET request to [aggregator]/hpke_config, where [aggregator] is the aggregator's endpoint URL, obtained from the task parameters. Clients **MAY** specify a query parameter task_id when sending an HTTP GET request to [aggregator]/hpke_config?task_id=[task-id], where [task-id] is the task ID obtained from the task parameters, encoded in Base 64 with URL and filename safe alphabet with no padding, as specified in sections 5 and 3.2 of [RFC4648]. If the aggregator does not recognize the task ID, then it **MUST** abort with error unrecognizedTask.

An aggregator is free to use different HPKE configurations for each task with which it is configured. If the task ID is missing from a client's request, the aggregator **MAY** abort with an error of type missingTaskID, in which case the client **SHOULD** retry the request with a well-formed task ID included.

An aggregator responds to well-formed requests with HTTP status code 200 OK and an HpkeConfigList value. The HpkeConfigList structure

contains one or more HpkeConfig structures in decreasing order of preference. This allows a server to support multiple HPKE configurations simultaneously.

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

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

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

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

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

Aggregators **SHOULD** allocate distinct id values for each HpkeConfig in a HpkeConfigList. The **RECOMMENDED** strategy for generating these values is via rejection sampling, i.e., to randomly select an id value repeatedly until it does not match any known HpkeConfig.

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

- the GET request failed or did not return a valid HPKE configuration; or
- the HPKE configuration specifies a KEM, KDF, or AEAD algorithm the client does not recognize.

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

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

Clients **SHOULD** follow the usual HTTP caching [RFC9111] semantics for key configurations.

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

4.3.2. Upload Request

Clients upload reports by using an HTTP POST to [leader]/upload, where [leader] is the first entry in the task's aggregator endpoints. The payload is structured as follows:

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

struct {
  TaskID task_id;
  ReportMetadata metadata;
  opaque public_share<0..2^32-1>;
  HpkeCiphertext encrypted_input_shares<1..2^32-1>;
} Report;
```

This message is called the Client's report. It consists of the task ID, report metadata, the "public share" output by the VDAF's input-distribution algorithm, and the encrypted input share of each of the Aggregators. (Note that the public share might be empty, depending on the VDAF. For example, Prio3 has an empty public share, but Poplar1 does not. See [VDAF].) The header consists of the task ID and report "metadata". The metadata consists of the following fields:

- A report ID used by the Aggregators to ensure the report appears in at most one batch. (See Section 4.4.1.4.) The Client **MUST** generate this by generating 16 random bytes using a cryptographically secure random number generator.

- A timestamp representing the time at which the report was generated. Specifically, the time field is set to the number of seconds elapsed since the start of the UNIX epoch. The client **SHOULD** round this value down to the nearest multiple of time_precision in order to ensure that that the timestamp cannot be used to link a report back to the Client that generated it.

To generate a report, the Client begins by sharding its measurement into input shares as specified by the VDAF. It then wraps each input share in the following structure:

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

Field extensions is set to the list of extensions intended to be consumed by the given Aggregator. (See Section 4.3.3.) Field payload is set to the Aggregator's input share output by the VDAF sharding algorithm.

Next, the Client encrypts each PlaintextInputShare plaintext_input_share as follows:

```
enc, payload = SealBase(pk,
  "dap-03 input share" || 0x01 || server_role,
  input_share_aad, plaintext_input_share)
```

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

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

The order of the encrypted input shares appear **MUST** match the order of the task's aggregator_endpoints. That is, the first share should be the leader's, the second share should be for the first helper, and so on.

The leader responds to well-formed requests to [leader]/upload with HTTP status code 200 OK. Malformed requests are handled as described in Section 3.2. Clients **SHOULD NOT** upload the same measurement value in more than one report if the leader responds with HTTP status code 200 OK.

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

The leader responds to requests whose leader encrypted input share uses an out-of-date or unknown HpkeConfig.id value, indicated by HpkeCiphertext.config_id, with error of type 'outdatedConfig'. If the leader supports multiple HPKE configurations, it can use trial

decryption with each configuration to determine if requests match a known HPKE configuration. When clients receive an 'outdatedConfig' error, they **SHOULD** invalidate any cached aggregator HpkeConfigList and retry with a freshly generated Report. If this retried report does not succeed, clients **SHOULD** abort and discontinue retrying.

If a report's ID matches that of a previously uploaded report, the leader **MUST** ignore it and the leader **MAY** alert the client with error reportRejected. See the implementation note in Section 4.4.1.4.

The Leader **MUST** ignore any report pertaining to a batch that has already been collected (see Section 4.4.1.4 for details). Otherwise, comparing the aggregate result to the previous aggregate result may result in a privacy violation. Note that this is enforced by all Aggregators, not just the leader. In addition, the leader **MAY** abort the upload protocol and alert the client with error reportRejected.

The Leader **MAY** ignore any reports whose timestamp is past the task's task_expiration. When it does so, the leader **SHOULD** abort the upload protocol and alert the client with error reportRejected. Client **MAY** choose to opt out of the task if its own clock has passed task_expiration.

Leaders can buffer reports while waiting to aggregate them. The leader **SHOULD NOT** accept reports whose timestamps are too far in the future. Implementors **MAY** provide for some small leeway, usually no more than a few minutes, to account for clock skew. If the leader rejects a report for this reason, it **SHOULD** abort the upload protocol and alert the client with error reportTooEarly. In this situation, clients **MAY** re-upload the report later on.

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

4.3.3. Upload Extensions

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

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

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

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

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

Extensions are mandatory-to-implement: If an Aggregator receives a Report containing an extension it does not recognize, then it **MUST** reject the Report. (See Section 4.4.1.4 for details.)

4.3.4. Upload Message Security

The contents of each input share must be kept confidential from everyone but the client and the aggregator it is being sent to. In addition, clients must be able to authenticate the aggregator they upload to.

HTTPS provides confidentiality between the DAP client and the leader, but this is not sufficient since the helper's report shares are relayed through the leader. Confidentiality of report shares is achieved by encrypting each report share to a public key held by the respective aggregator using [HPKE]. Clients fetch the public keys from each aggregator over HTTPS, allowing them to authenticate the server.

Aggregators **MAY** require clients to authenticate when uploading reports. This is an effective mitigation against Sybil [Dou02] attacks in deployments where it is practical for each client to have an identity provisioned (e.g., a user logged into an online service or a hardware device programmed with an identity). If it is used, client authentication **MUST** use a scheme that meets the requirements in Section 3.1.

In some deployments, it will not be practical to require clients to authenticate (e.g., a widely distributed application that does not require its users to login to any service), so client authentication is not mandatory in DAP.

[[OPEN ISSUE: deployments that don't have client auth will need to do something about Sybil attacks. Is there any useful guidance or **SHOULD** we can provide? Sort of relevant: issue #89]]

4.4. Verifying and Aggregating Reports

Once a set of clients have uploaded their reports to the leader, the leader can send them to the helpers to be verified and aggregated. In order to enable the system to handle very large batches of reports, this process can be performed incrementally. Verification of a set of reports is referred to as an aggregation job. Each aggregation job is associated with exactly one DAP task, and a DAP task can have many aggregation jobs. Each job is associated with an ID that is unique within the context of a DAP task in order to distinguish different jobs from one another. Each aggregator uses this ID as an index into per-job storage, e.g., to keep track of report shares that belong to a given aggregation job.

To run an aggregation job, the leader sends a message to each helper containing the report shares in the job. The helper then processes them (verifying the proofs and incorporating their values into the ongoing aggregate) and replies to the leader.

The exact structure of the aggregation job flow depends on the VDAF. Specifically:

* Some VDAFs (e.g., Prio3) allow the leader to start aggregating reports proactively before all the reports in a batch are received. Others (e.g., Poplar1) require all the reports to be present and must be initiated by the collector.

* Processing the reports -- especially validating them -- may require multiple round trips.

Note that it is possible to aggregate reports from one batch while reports from the next batch are coming in. This is because each report is validated independently.

This process is illustrated below in Figure 2. In this example, the batch size is 20, but the leader opts to process the reports in sub-batches of 10. Each sub-batch takes two round-trips to process.

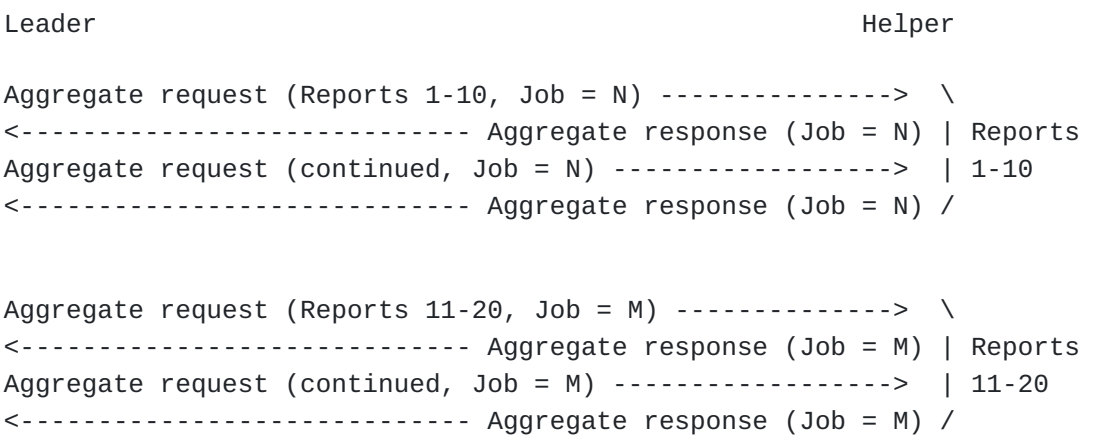

Figure 2: Aggregation Flow (batch size=20). Multiple aggregation flows can be executed at the same time.

[OPEN ISSUE: Should there be an indication of whether a given aggregate request is a continuation of a previous sub-batch?]

The aggregation flow can be thought of as having three phases for transforming each valid input report share into an output share:

- Initialization: Begin the aggregation flow by sharing report shares with each helper. Each aggregator, including the leader, initializes the underlying VDAF instance using these report shares and the VDAF configured for the corresponding measurement task.

- Continuation: Continue the aggregation flow by exchanging messages produced by the underlying VDAF instance until aggregation completes or an error occurs. These messages do not replay the shares.

- Completion: Finish the aggregate flow, yielding an output share corresponding for each input report share in the batch.

4.4.1. Aggregate Initialization

The leader begins an aggregation job by choosing a set of candidate reports that pertain to the same DAP task and a unique job ID. The job ID is a 32-byte value, structured as follows:

```
opaque AggregationJobID[32];
```

The leader can run this process for many candidate reports in parallel as needed. After choosing the set of candidates, the leader begins aggregation by splitting each report into "report shares",

one for each aggregator. The leader and helpers then run the aggregate initialization flow to accomplish two tasks:

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

2. For each valid report share, initialize the VDAF preparation process.

An invalid report share is marked with one of the following errors:

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

The leader and helper initialization behavior is detailed below.

4.4.1.1. Leader Initialization

The leader begins the aggregate initialization phase with the set of candidate report shares as follows:

1. Generate a fresh AggregationJobID. This ID **MUST** be unique within the context of the corresponding DAP task. It is **RECOMMENDED** that this be set to a random string output by a cryptographically secure pseudorandom number generator.

2. Decrypt the input share for each report share as described in Section 4.4.1.3.

3. Check that the resulting input share is valid as described in Section 4.4.1.4.

4. Initialize VDAF preparation as described in Section 4.4.1.5.

If any step yields an invalid report share, the leader removes the report share from the set of candidate reports. Once the leader has initialized this state for all valid candidate report shares, it then creates an AggregateInitializeReq message for each helper to initialize the preparation of this candidate set. The AggregateInitializeReq message is structured as follows:

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

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

struct {
  TaskID task_id;
  AggregationJobID job_id;
  opaque agg_param<0..2^32-1>;
  PartialBatchSelector part_batch_selector;
  ReportShare report_shares<1..2^32-1>;
} AggregateInitializeReq;
```

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

This message consists of:

* The ID of the task for which the reports will be aggregated.

* The aggregation job ID.

* The opaque, VDAF-specific aggregation parameter provided during the collection flow (agg_param),

  [[OPEN ISSUE: Check that this handling of agg_param is appropriate when the definition of Poplar is done.]]

* Information used by the Aggregators to determine how to aggregate each report:

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

    [OPEN ISSUE: For fixed_size tasks, the Leader is in complete control over which batch a report is included in. For time_interval tasks, the Client has some control, since the timestamp determines which batch window it falls in. Is this

desirable from a privacy perspective? If not, it might be simpler to drop the timestamp altogether and have the agg init request specify the batch window instead.]

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

*The sequence of report shares to aggregate. The encrypted_input_share field of the report share is the HpkeCiphertext whose index in Report.encrypted_input_shares is equal to the index of the aggregator in the task's aggregator_endpoints to which the AggregateInitializeReq is being sent.

Let [aggregator] denote the helper's API endpoint. The leader sends a POST request to [aggregator]/aggregate with its AggregateInitializeReq message as the payload. The media type is "application/dap-aggregate-initialize-req".

4.4.1.2. Helper Initialization

Each helper begins their portion of the aggregate initialization phase with the set of candidate report shares obtained in an AggregateInitializeReq message from the leader. It attempts to recover and validate the corresponding input shares similar to the leader, and eventually returns a response to the leader carrying a VDAF-specific message for each report share.

To begin this process, the helper first checks if it recognizes the task ID, if not then it **MUST** abort with error unrecognizedTask. Then the helper checks that the report IDs in AggregateInitializeReq.report_shares are all distinct. If two ReportShare values have the same report ID, then the helper **MUST** abort with error unrecognizedMessage. If this check succeeds, the helper then attempts to recover each input share in AggregateInitializeReq.report_shares as follows:

1. Decrypt the input share for each report share as described in Section 4.4.1.3.

2. Check that the resulting input share is valid as described in Section 4.4.1.4.

3. Initialize VDAF preparation and initial outputs as described in Section 4.4.1.5.

[[OPEN ISSUE: consider moving the helper report ID check into Section 4.4.1.4]]

Once the helper has processed each valid report share in AggregateInitializeReq.report_shares, the helper then creates an AggregateInitializeResp message to complete its initialization. This message is structured as follows:

```
enum {
  continued(0),
  finished(1),
  failed(2),
  (255)
} PrepareStepResult;

struct {
  ReportID report_id;
  PrepareStepResult prepare_step_result;
  select (PrepareStep.prepare_step_result) {
    case continued: opaque prep_msg<0..2^32-1>; /* VDAF preparation mess
    case finished:  Empty;
    case failed:    ReportShareError;
  };
} PrepareStep;

struct {
  PrepareStep prepare_steps<1..2^32-1>;
} AggregateInitializeResp;
```

The message is a sequence of PrepareStep values, the order of which matches that of the ReportShare values in AggregateInitializeReq.report_shares. Each report that was marked as invalid is assigned the PrepareStepResult failed. Otherwise, the PrepareStep is either marked as continued with the output prep_msg, or is marked as finished if the VDAF preparation process is finished for the report share.

The helper's response to the leader is an HTTP status code 200 OK whose body is the AggregateInitializeResp and media type is "application/dap-aggregate-initialize-resp".

Upon receipt of a helper's AggregateInitializeResp message, the leader checks that the sequence of PrepareStep messages corresponds to the ReportShare sequence of the AggregateInitializeReq. If any message appears out of order, is missing, has an unrecognized report ID, or if two messages have the same report ID, then the leader **MUST** abort with error "unrecognizedMessage".

[[OPEN ISSUE: the leader behavior here is sort of bizarre -- to whom does it abort?]]

4.4.1.3. Input Share Decryption

Each report share has a corresponding task ID, report metadata (report ID and, timestamp), the public share sent to each Aggregator, and the recipient's encrypted input share. Let task_id, 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, metadata, and public_share. Let this be denoted by input_share_aad. Then, the aggregator looks up the HPKE config and corresponding secret key indicated by encrypted_input_share.config_id and attempts decryption of the payload with the following procedure:

```
plaintext_input_share = OpenBase(encrypted_input_share.enc, sk,
  "dap-03 input share" || 0x01 || server_role,
  input_share_aad, encrypted_input_share.payload)
```

where sk is the HPKE secret key, and server_role is the role of the 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 leader supports multiple HPKE configurations with non-distinct configuration identifiers, it can use trial decryption with each configuration. If decryption fails, the aggregator marks the report share as invalid with the error hpke_decrypt_error. Otherwise, the aggregator outputs the resulting PlaintextInputShare plaintext_input_share.

4.4.1.4. Input Share Validation

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

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

1. Check that the decrypted input share can be decoded. If not, the input share **MUST** be marked as invalid with the error unrecognized_message.

2. Check if the report has already been aggregated with this aggregation parameter. If this check fails, the input share **MUST** be marked as invalid with the error report_replayed. This is the case if the report was used in a previous aggregate request and is therefore a replay.

   *Implementation note: To detect replay attacks, each Aggregator is required to keep track of the set of reports

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

3. Check if the report has never been aggregated but is contained by a batch that has been collected. If this check fails, the input share **MUST** be marked as invalid with the error batch_collected. This prevents additional reports from being aggregated after its batch has already been collected.

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

   * For fixed_size tasks, the Aggregators need to ensure that each batch is roughly the same size. If the number of reports aggregated for the current batch exceeds the maximum batch size (per the task configuration), the Aggregator **MAY** mark the input share as invalid with the error "batch_saturated". Note that this behavior is not strictly enforced here but during the collect sub-protocol. (See Section 4.5.6.) If both checks succeed, the input share is not marked as invalid.

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

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

7. Check if the PlaintextInputShare contains unrecognized extensions. If so, then the Aggregator **MUST** mark the input share as invalid with error "unrecognized_message".

8. Check if the ExtensionType of any two extensions in PlaintextInputShare is the same. If so, then the Aggregator **MUST** mark the input share as invalid with error "unrecognized_message".

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

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

4.4.1.5. Input Share Preparation

Input share preparation consists of running the preparation-state initialization algorithm for the VDAF associated with the task and computes the first state transition. This produces three possible values:

1. An error, in which case the input report share is marked as invalid.

2. An output share, in which case the aggregator stores the output share for future collection as described in Section 4.5.

3. An initial VDAF state and preparation message, denoted (prep_state, prep_msg).

Each aggregator runs this procedure for a given input share with corresponding report ID as follows:

```
prep_state = VDAF.prep_init(vdaf_verify_key,
                            agg_id,
                            agg_param,
                            report_id,
                            public_share,
                            plaintext_input_share.payload)
out = VDAF.prep_next(prep_state, None)
```

vdaf_verify_key is the VDAF verification key shared by the aggregators; agg_id is the aggregator ID (0x00 for the Leader and 0x01 for the helper); agg_param is the opaque aggregation parameter distributed to the aggregators by the collector; public_share is the public share generated by the client and distributed to each aggregator; and plaintext_input_share is the aggregator's PlaintextInputShare.

If either step fails, the aggregator marks the report as invalid with error vdaf_prep_error. Otherwise, the value out is interpreted as follows. If this is the last round of the VDAF, then out is the aggregator's output share. Otherwise, out is the pair (prep_state, prep_msg).

4.4.2. Aggregate Continuation

In the continuation phase, the leader drives the VDAF preparation of each share in the candidate report set until the underlying VDAF moves into a terminal state, yielding an output share for all aggregators or an error. This phase may involve multiple rounds of

interaction depending on the underlying VDAF. Each round trip is initiated by the leader.

4.4.2.1. Leader Continuation

The leader begins each round of continuation for a report share based on its locally computed prepare message and the previous PrepareStep from the helper. If PrepareStep is of type failed, then the leader acts based on the value of the ReportShareError:

- If the error is ReportShareError.report_too_early, then the leader **MAY** try to re-send the report in a later AggregateInitializeReq.

- For any other error, the leader marks the report as failed, removes it from the candidate report set and does not process it further.

If the type is finished, then the leader aborts with unrecognizedMessage. If the type is continued, then the leader proceeds as follows.

Let leader_outbound denote the leader's prepare message and helper_outbound denote the helper's. The leader computes the next state transition as follows:

```
inbound = VDAF.prep_shares_to_prep(agg_param, [leader_outbound, helper_o
out = VDAF.prep_next(prep_state, inbound)
```

where [leader_outbound, helper_outbound] is a vector of two elements. If either of these operations fails, then the leader marks the report as invalid. Otherwise it interprets out as follows: If this is the last round of the VDAF, then out is the aggregator's output share, in which case the aggregator finishes and stores its output share for further processing as described in Section 4.5. Otherwise, out is the pair (new_state, prep_msg), where new_state is its updated state and prep_msg is its next VDAF message (which will be leader_outbound in the next round of continuation). For the latter case, the helper sets prep_state to new_state.

The leader then sends each PrepareStep to the helper in an AggregateContinueReq message, structured as follows:

```
struct {
  TaskID task_id;
  AggregationJobID job_id;
  PrepareStep prepare_steps<1..2^32-1>;
} AggregateContinueReq;
```

For each aggregator endpoint [aggregator] in AggregateContinueReq.task_id's parameters except its own, the leader sends a POST request to [aggregator]/aggregate with AggregateContinueReq as the payload and the media type set to "application/dap-aggregate-continue-req".

4.4.2.2. Helper Continuation

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

Otherwise, the helper continues with preparation for a report share by combining the leader's input message in AggregateContinueReq and its current preparation state (prep_state). This step yields one of three outputs:

1. An error, in which case the input report share is marked as invalid.

2. An output share, in which case the helper stores the output share for future collection as described in Section 4.5.

3. An updated VDAF state and preparation message, denoted (prep_state, prep_msg).

To carry out this step, for each PrepareStep in AggregateContinueReq.prepare_steps received from the leader, the helper performs the following check to determine if the report share should continue being prepared:

* If failed, then mark the report as failed and reply with a failed PrepareStep to the leader.

* If finished, then mark the report as finished and reply with a finished PrepareStep to the leader. The helper then stores the output share and awaits for collection; see Section 4.5.

Otherwise, preparation continues. In this case, the helper computes its updated state and output message as follows:

```
out = VDAF.prep_next(prep_state, inbound)
```

where inbound is the previous VDAF prepare message sent by the leader and prep_state is the helper's current preparation state. If this operation fails, then the helper fails with error vdaf_prep_error. Otherwise, it interprets out as follows:

* If this is the last round of VDAF preparation phase, then out is the helper's output share, in which case the helper stores the output share for future collection.

*Otherwise, the helper interprets out as the tuple (new_state, prep_msg), where new_state is its updated preparation state and prep_msg is its next VDAF message.

This output message for each report in AggregateContinueReq.prepare_steps is then sent to the leader in an AggregateContinueResp message, structured as follows:

```
struct {
  PrepareStep prepare_steps<1..2^32-1>;
} AggregateContinueResp;
```

The order of AggregateContinueResp.prepare_steps **MUST** match that of the PrepareStep values in AggregateContinueReq.prepare_steps. The helper's response to the leader is an HTTP status code 200 OK whose body is the AggregateContinueResp and media type is "application/dap-aggregate-continue-resp". The helper then awaits the next message from the leader.

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

4.4.3. Aggregate Message Security

Aggregate sub-protocol messages must be confidential and mutually authenticated.

The aggregate sub-protocol is driven by the leader acting as an HTTPS client, making requests to the helper's HTTPS server. HTTPS provides confidentiality and authenticates the helper to the leader.

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

4.5. Collecting Results

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

1. Collect request and response between the collector and leader, specified in Section 4.5.1

2. Aggregate share request and response between the leader and each aggregator, specified in Section 4.5.2

Once complete, the collector computes the final aggregate result as specified in Section 4.5.3.

4.5.1. Collection Initialization

To initiate collection, the collector issues a POST request to [leader]/collect, where [leader] is the leader's endpoint URL. The body of the request is structured as follows:

[OPEN ISSUE: Decide if and how the collector's request is authenticated. If not, then we need to ensure that collect job URIs are resistant to enumeration attacks.]

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

The named parameters are:

- task_id, the DAP task ID. If the Leader does not recognize the task ID, it **MUST** abort with error unrecognizedTask.
- query, the Collector's query. The indicated query type **MUST** match the task's query type. Otherwise, the Leader **MUST** abort with error "queryMismatch".
- agg_param, an aggregation parameter for the VDAF being executed. This is the same value as in AggregateInitializeReq (see Section 4.4.1.1).

Depending on the VDAF scheme and how the leader is configured, the leader and helper may already have prepared a sufficient number of reports satisfying the query and be ready to return the aggregate shares right away, but this cannot be guaranteed. In fact, for some VDAFs, it is not be possible to begin preparing inputs until the collector provides the aggregation parameter in the CollectReq. For these reasons, collect requests are handled asynchronously.

Upon receipt of a CollectReq, the leader begins by checking that the request meets the requirements of the batch parameters using the procedure in Section 4.5.6. If so, it immediately sends the collector a response with HTTP status 303 See Other and a Location header containing a URI identifying the collect job that can be polled by the collector, called the "collect job URI".

The leader then begins working with the helper to prepare the shares satisfying the query (or continues this process, depending on the VDAF) as described in Section 4.4.

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

If the leader has not yet obtained an aggregator share from each aggregator, the leader invokes the aggregate share request flow described in Section 4.5.2. Otherwise, when all aggregator shares are successfully obtained, the leader responds to subsequent HTTP GET requests to the collect job's URI with HTTP status code 200 OK and a body consisting of a CollectResp:

```
struct {
  PartialBatchSelector part_batch_selector;
  uint64 report_count;
  HpkeCiphertext encrypted_agg_shares<1..2^32-1>;
} CollectResp;
```

This structure includes the following:

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

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

- The number of reports included in the batch.

- The vector of encrypted aggregate shares. They **MUST** appear in the same order as the aggregator endpoints list of the task parameters.

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

The collector can send an HTTP DELETE request to the collect job URI, to which the leader **MUST** respond with HTTP status 204 No Content. The leader **MAY** respond with HTTP status 204 No Content for requests to a collect job URI which has not received a DELETE request, for example if the results have been deleted due to age. The leader **MUST** respond to subsequent requests to the collect job URI with HTTP status 204 No Content.

[OPEN ISSUE: Describe how intra-protocol errors yield collect errors (see issue#57). For example, how does a leader respond to a collect request if the helper drops out?]

4.5.2. Collection Aggregation

The leader obtains each helper's encrypted aggregate share in order to respond to the collector's collect response. To do this, the leader first computes a checksum over the set of output shares included in the batch. The checksum is computed by taking the SHA256 [SHS] hash of each report ID from the client reports included in the aggregation, then combining the hash values with a bitwise-XOR operation.

Then, for each aggregator endpoint [aggregator] in the parameters associated with CollectReq.task_id (see Section 4.5) except its own, the leader sends a POST request to [aggregator]/aggregate_share with the following message:

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

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

The message contains the following parameters:

- The DAP task ID. If the Helper does not recognize the task ID, it **MUST** abort with error unrecognizedTask.

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

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

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

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

*The opaque aggregation parameter for the VDAF being executed. This value **MUST** match the same value in the the AggregateInitializeReq message sent in at least one run of the aggregate sub-protocol. (See Section 4.4.1.1). and in CollectReq (see Section 4.5.1).

*The number number of reports included in the batch.

*The batch checksum.

To handle the leader's request, the helper first ensures that the request meets the requirements for batch parameters following the procedure in Section 4.5.6.

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

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

```
agg_share = VDAF.out_shares_to_agg_share(agg_param, out_shares)
```

Note that for most VDAFs, it is possible to aggregate output shares as they arrive rather than wait until the batch is collected. To do so however, it is necessary to enforce the batch parameters as described in Section 4.5.6 so that the aggregator knows which aggregate share to update.

The helper then encrypts agg_share under the collector's HPKE public key as described in Section 4.5.4, yielding encrypted_agg_share. Encryption prevents the leader from learning the actual result, as it only has its own aggregate share and cannot compute the helper's.

The helper responds to the leader with HTTP status code 200 OK and a body consisting of an AggregateShareResp:

```
struct {
  HpkeCiphertext encrypted_aggregate_share;
} AggregateShareResp;
```

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 uses the HpkeCiphertext to respond to a collect request (see Section 4.5).

After issuing an aggregate-share request for a given query, it is an error for the leader to issue any more aggregation jobs for additional reports that satisfy the query. These reports will be rejected by helpers as described Section 4.4.1.

Before completing the collect request, the leader also computes its own aggregate share agg_share by aggregating all of the prepared output shares that fall within the batch interval. Finally, it encrypts it under the collector's HPKE public key as described in Section 4.5.4.

4.5.3. Collection Finalization

Once the collector has received a successful collect response from the leader, it can decrypt the aggregate shares and produce an aggregate result. The collector decrypts each aggregate share as described in Section 4.5.4. If the collector successfully decrypts all aggregate shares, the collector then unshards the aggregate shares into an aggregate result using the VDAF's agg_shares_to_result algorithm. In particular, let agg_shares denote the ordered sequence of aggregator shares, ordered by aggregator index, let report_count denote the report count sent by the Leader, and let agg_param be the opaque aggregation parameter. The final aggregate result is computed as follows:

```
agg_result = VDAF.agg_shares_to_result(agg_param,
                                       agg_shares,
                                       report_count)
```

4.5.4. Aggregate Share Encryption

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

```
enc, payload = SealBase(pk, "dap-03 aggregate share" || server_role || 0
  agg_share_aad, agg_share)
```

where pk is the HPKE public key encoded by the collector's HPKE key, server_role is 0x02 for the leader and 0x03 for a helper, and agg_share_aad is a value of type AggregateShareAad with its values set from the corresponding fields of the AggregateShareReq. The SealBase() function is as specified in [HPKE], Section 6.1 for the ciphersuite indicated by the HPKE configuration.

```
struct {
  TaskID task_id;
  BatchSelector batch_selector;
} AggregateShareAad;
```

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-03 aggregate share" ||
  server_role || 0x00, agg_share_aad, enc_share.payload)
```

where sk is the HPKE secret key, server_role is the role of the server that sent the aggregate share (0x02 for the leader and 0x03 for the helper), and agg_share_aad is an AggregateShareAad message constructed from the task ID in the collect request and a batch selector. The value of the batch selector used in agg_share_aad is computed by the Collector from its query and the response to its query as follows:

- For time_interval tasks, the batch selector is the batch interval specified in the query.
- For fixed_size tasks, the batch selector is the batch ID assigned sent in the response.

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

4.5.5. Collect Message Security

Collect sub-protocol messages must be confidential and mutually authenticated.

HTTPS provides confidentiality and authenticates the leader to the collector. Additionally, the leader encrypts its aggregate share to a public key held by the collector using [HPKE].

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

[[OPEN ISSUE: collector public key is currently in the task parameters, but this will have to change #102]]

The collector and helper never directly communicate with each other, but the helper does transmit an aggregate share to the collector through the leader, as detailed in Section 4.5.2. The aggregate share must be confidential from everyone but the helper and the collector.

Confidentiality is achieved by having the helper encrypt its aggregate share to a public key held by the collector using [HPKE].

There is no authentication between the collector and the helper. This allows the leader to:

* Send collect parameters to the helper that do not reflect the parameters chosen by the collector

* Discard the aggregate share computed by the helper and then fabricate aggregate shares that combine into an arbitrary aggregate result

These are attacks on robustness, which we already assume to hold only if both aggregators are honest, which puts these malicious-leader attacks out of scope (see Section 7).

[[OPEN ISSUE: Should we have authentication in either direction between the helper and the collector? #155]]

4.5.6. Batch Validation

Before an Aggregator responds to a CollectReq or AggregateShareReq, it must first check that the request does not violate the parameters associated with the DAP task. It does so as described here.

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

Next, the Aggregator checks that batch contains a valid number of reports, as determined by the query type. If the size check fails, then the Aggregator **MUST** abort with error of type "invalidBatchSize".

Next, the Aggregator checks that the batch has not been aggregated too many times. This is determined by the maximum number of times a batch can be queried, max_batch_query_count. Unless the query has been issued less than max_batch_query_count times, the Aggregator **MUST** abort with error of type "batchQueriedTooManyTimes".

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

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

4.5.6.1. Time-interval Queries

4.5.6.1.1. Boundary Check

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

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

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

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

4.5.6.1.2. Size Check

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

4.5.6.2. Fixed-size Queries

4.5.6.2.1. Boundary Check

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

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

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

4.5.6.2.2. Size Check

The query configuration specifies the minimum batch size, min_batch_size, and maximum batch size, max_batch_size. The Aggregator checks that len(X) >= min_batch_size and len(X) <= max_batch_size, where X is the set of reports in the batch.

5. Operational Considerations

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

5.1. Protocol participant capabilities

The design in this document has different assumptions and requirements for different protocol participants, including clients, aggregators, and collectors. This section describes these capabilities in more detail.

5.1.1. Client capabilities

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

5.1.2. Aggregator capabilities

Helpers and leaders 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 sub-protocol, which includes validating and aggregating reports; and

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

In addition, for each DAP task, helpers are required to:

- Implement some form of batch-to-report index, as well as inter- and intra-batch replay mitigation storage, which includes some way of tracking batch report size. Some of this state may be used for replay attack mitigation. The replay mitigation strategy is described in Section 4.4.1.4.

Beyond the minimal capabilities required of helpers, leaders are generally required to:

- Support the upload protocol and store reports; and
- Track batch report size during each collect flow and request encrypted output shares from helpers.

In addition, for each DAP task, leaders are required to:

- Implement and store state for the form of inter- and intra-batch replay mitigation in Section 4.4.1.4.

5.1.3. Collector capabilities

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

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

5.2. Data resolution limitations

Privacy comes at the cost of computational complexity. While affine-aggregatable encodings (AFEs) can compute many useful statistics, they require more bandwidth and CPU cycles to account for finite-field arithmetic during input-validation. The increased work from verifying inputs decreases the throughput of the system or the inputs processed per unit time. Throughput is related to the verification circuit's complexity and the available compute-time to each aggregator.

Applications that utilize proofs with a large number of multiplication gates or a high frequency of inputs may need to limit inputs into the system to meet bandwidth or compute constraints. Some methods of overcoming these limitations include choosing a better representation for the data or introducing sampling into the data collection methodology.

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

5.3. Aggregation utility and soft batch deadlines

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

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

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

5.4. Protocol-specific optimizations

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

5.4.1. Reducing storage requirements

In general, the aggregators are required to keep state for tasks and all valid reports for as long as collect requests can be made for them. In particular, aggregators must store a batch as long as the batch has not been queried more than max_batch_query_count times. However, it is not always necessary to store the reports themselves. For schemes like Prio3 [VDAF] in which reports are verified only once, each aggregator only needs to store its aggregate share for each possible batch interval, along with the number of times the aggregate share was used in a batch. This is due to the requirement that the batch interval respect the boundaries defined by the DAP parameters. (See Section 4.5.6.)

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 4.4.1.4. Aggregators **SHOULD** take steps to mitigate the risk of dropping reports (e.g., by evicting the oldest data first).

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

6. Compliance Requirements

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

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

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

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

7. Security Considerations

DAP assumes an active attacker that controls the network and has the ability to statically corrupt any number of clients, aggregators, and collectors. That is, the attacker can learn the secret state of any party prior to the start of its attack. For example, it may coerce a client into providing malicious input shares for aggregation or coerce an aggregator into diverting from the protocol specified (e.g., by divulging its input shares to the attacker).

In the presence of this adversary, DAP aims to achieve the privacy and robustness security goals described in [VDAF]'s Security Considerations section.

Currently, the specification does not achieve these goals. In particular, there are several open issues that need to be addressed before these goals are met. Details for each issue are below.

1. When crafted maliciously, collect requests may leak more information about the measurements than the system intends. For example, the spec currently allows sequences of collect requests to reveal an aggregate result for a batch smaller than the minimum batch size. [OPEN ISSUE: See issue#195. This also has implications for how we solve issue#183.]

2. Even benign collect requests may leak information beyond what one might expect intuitively. For example, the Poplar1 VDAF [VDAF] can be used to compute the set of heavy hitters among a set of arbitrary bit strings uploaded by clients. This requires multiple evaluations of the VDAF, the results of which reveal information to the aggregators and collector beyond what follows from the heavy hitters themselves. Note that this

leakage can be mitigated using differential privacy. [OPEN ISSUE: We have yet not specified how to add DP.]

3. The core DAP spec does not defend against Sybil attacks. 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: 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"). The upload sub-protocol includes an extensions mechanism that can be used to prevent --- or at least mitigate --- these types of attacks. See Section 4.3.3. [OPEN ISSUE: No such extension has been implemented, so we're not yet sure if the current mechanism is sufficient.]

7.1. Threat model

[OPEN ISSUE: This subsection is a bit out-of-date.]

In this section, we enumerate the actors participating in the Prio system and enumerate their assets (secrets that are either inherently valuable or which confer some capability that enables further attack on the system), the capabilities that a malicious or compromised actor has, and potential mitigations for attacks enabled by those capabilities.

This model assumes that all participants have previously agreed upon and exchanged all shared parameters over some unspecified secure channel.

7.1.1. Client/user

7.1.1.1. Assets

1. Unshared inputs. Clients are the only actor that can ever see the original inputs.

2. Unencrypted input shares.

7.1.1.2. Capabilities

1. Individual users can reveal their own input and compromise their own privacy.

2. Clients (that is, software which might be used by many users of the system) can defeat privacy by leaking input outside of the Prio system.

3. Clients may affect the quality of aggregations by reporting false input.

    *Prio can only prove that submitted input is valid, not that it is true. False input can be mitigated orthogonally to the Prio protocol (e.g., by requiring that aggregations include a minimum number of contributions) and so these attacks are considered to be outside of the threat model.

4. Clients can send invalid encodings of input.

7.1.1.3. Mitigations

1. The input validation protocol executed by the aggregators prevents either individual clients or a coalition of clients from compromising the robustness property.

2. If aggregator output satisfies differential privacy Section 7.5, then all records not leaked by malicious clients are still protected.

7.1.2. Aggregator

7.1.2.1. Assets

1. Unencrypted input shares.

2. Input share decryption keys.

3. Client identifying information.

4. Aggregate shares.

5. Aggregator identity.

7.1.2.2. Capabilities

1. Aggregators may defeat the robustness of the system by emitting bogus output shares.

2. If clients reveal identifying information to aggregators (such as a trusted identity during client authentication), aggregators can learn which clients are contributing input.

    1. Aggregators may reveal that a particular client contributed input.

2. Aggregators may attack robustness by selectively omitting inputs from certain clients.

    *For example, omitting submissions from a particular geographic region to falsely suggest that a particular localization is not being used.

3. Individual aggregators may compromise availability of the system by refusing to emit aggregate shares.

4. Input validity proof forging. Any aggregator can collude with a malicious client to craft a proof that will fool honest aggregators into accepting invalid input.

5. Aggregators can count the total number of input shares, which could compromise user privacy (and differential privacy Section 7.5) if the presence or absence of a share for a given user is sensitive.

7.1.2.3. Mitigations

1. The linear secret sharing scheme employed by the client ensures that privacy is preserved as long as at least one aggregator does not reveal its input shares.

2. If computed over a sufficient number of reports, aggregate shares reveal nothing about either the inputs or the participating clients.

3. Clients can ensure that aggregate counts are non-sensitive by generating input independently of user behavior. For example, a client should periodically upload a report even if the event that the task is tracking has not occurred, so that the absence of reports cannot be distinguished from their presence.

4. Bogus inputs can be generated that encode "null" shares that do not affect the aggregate output, but mask the total number of true inputs.

    *Either leaders or clients can generate these inputs to mask the total number from non-leader aggregators or all the aggregators, respectively.

    *In either case, care must be taken to ensure that bogus inputs are indistinguishable from true inputs (metadata, etc), especially when constructing timestamps on reports.

[OPEN ISSUE: Define what "null" shares are. They should be defined such that inserting null shares into an aggregation is effectively a no-op. See issue#98.]

7.1.3. Leader

The leader is also an aggregator, and so all the assets, capabilities and mitigations available to aggregators also apply to the leader.

7.1.3.1. Capabilities

1. Input validity proof verification. The leader can forge proofs and collude with a malicious client to trick aggregators into aggregating invalid inputs.

    *This capability is no stronger than any aggregator's ability to forge validity proof in collusion with a malicious client.

2. Relaying messages between aggregators. The leader can compromise availability by dropping messages.

    *This capability is no stronger than any aggregator's ability to refuse to emit aggregate shares.

3. Shrinking the anonymity set. The leader instructs aggregators to construct output parts and so could request aggregations over few inputs.

7.1.3.2. Mitigations

1. Aggregators enforce agreed upon minimum aggregation thresholds to prevent deanonymizing.

2. If aggregator output satisfies differential privacy Section 7.5, then genuine records are protected regardless of the size of the anonymity set.

7.1.4. Collector

7.1.4.1. Capabilities

1. Advertising shared configuration parameters (e.g., minimum thresholds for aggregations, joint randomness, arithmetic circuits).

2. Collectors may trivially defeat availability by discarding aggregate shares submitted by aggregators.

3. Known input injection. Collectors may collude with clients to send known input to the aggregators, allowing collectors to shrink the effective anonymity set by subtracting the known

inputs from the final output. Sybil attacks [Dou02] could be used to amplify this capability.

7.1.4.2. Mitigations

1. Aggregators should refuse shared parameters that are trivially insecure (i.e., aggregation threshold of 1 contribution).

2. If aggregator output satisfies differential privacy Section 7.5, then genuine records are protected regardless of the size of the anonymity set.

7.1.5. Aggregator collusion

If all aggregators collude (e.g. by promiscuously sharing unencrypted input shares), then none of the properties of the system hold. Accordingly, such scenarios are outside of the threat model.

7.1.6. Attacker on the network

We assume the existence of attackers on the network links between participants.

7.1.6.1. Capabilities

1. Observation of network traffic. Attackers may observe messages exchanged between participants at the IP layer.

    1. The time of transmission of input shares by clients could reveal information about user activity.

        * For example, if a user opts into a new feature, and the client immediately reports this to aggregators, then just by observing network traffic, the attacker can infer what the user did.

    2. Observation of message size could allow the attacker to learn how much input is being submitted by a client.

        * For example, if the attacker observes an encrypted message of some size, they can infer the size of the plaintext, plus or minus the cipher block size. From this they may be able to infer which aggregations the user has opted into or out of.

2. Tampering with network traffic. Attackers may drop messages or inject new messages into communications between participants.

7.1.6.2. Mitigations

1. All messages exchanged between participants in the system should be encrypted.

2. All messages exchanged between aggregators, the collector and the leader should be mutually authenticated so that network attackers cannot impersonate participants.

3. Clients should be required to submit inputs at regular intervals so that the timing of individual messages does not reveal anything.

4. Clients should submit dummy inputs even for aggregations the user has not opted into.

[[OPEN ISSUE: The threat model for Prio --- as it's described in the original paper and [BBCGGI19] --- considers **either** a malicious client (attacking robustness) **or** a malicious subset of aggregators (attacking privacy). In particular, robustness isn't guaranteed if any one of the aggregators is malicious; in theory it may be possible for a malicious client and aggregator to collude and break robustness. Is this a contingency we need to address? There are techniques in [BBCGGI19] that account for this; we need to figure out if they're practical.]]

7.2. Client authentication or attestation

[TODO: Solve issue#89]

7.3. Anonymizing proxies

Client reports can contain auxiliary information such as source IP, HTTP user agent or in deployments which use it, client authentication information, which could be used by aggregators to identify participating clients or permit some attacks on robustness. This auxiliary information could be removed by having clients submit reports to an anonymizing proxy server which would then use Oblivious HTTP [I-D.thomson-http-oblivious] to forward inputs to the DAP leader, without requiring any server participating in DAP to be aware of whatever client authentication or attestation scheme is in use.

7.4. Batch parameters

An important parameter of a DAP deployment is the minimum batch size. If an aggregation includes too few inputs, then the outputs can reveal information about individual participants. Aggregators use the batch size field of the shared task parameters to enforce minimum batch size during the collect protocol, but server

implementations may 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 minimum batch sizes.

The DAP parameters also specify the maximum number of times a report can be used. Some protocols, such as Poplar [BBCGGI21], require reports to be used in multiple batches spanning multiple collect requests.

7.5. Differential privacy

Optionally, DAP deployments can choose to ensure their output F achieves 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 outputs. 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 input in a batch except for one, that one record is still formally protected.

[OPEN ISSUE: While parameters configuring the differential privacy noise (like specific distributions / variance) can be agreed upon out of band by the aggregators and collector, there may be benefits to adding explicit protocol support by encoding them into task parameters.]

7.6. Robustness in the presence of malicious servers

Most DAP protocols, including Prio and Poplar, are robust against malicious clients, but are not robust against malicious servers. Any aggregator can simply emit bogus aggregate shares and undetectably spoil aggregates. If enough aggregators were available, this could be mitigated by running the protocol multiple times with distinct subsets of aggregators chosen so that no aggregator appears in all subsets and checking all the outputs against each other. If all the protocol runs do not agree, then participants know that at least one aggregator is defective, and it may be possible to identify the defector (i.e., if a majority of runs agree, and a single aggregator appears in every run that disagrees). See #22 for discussion.

7.7. Infrastructure diversity

Prio deployments should ensure that aggregators do not have common dependencies that would enable a single vendor to reassemble inputs. 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.

7.8. System requirements

7.8.1. Data types

8. IANA Considerations

8.1. Protocol Message Media Types

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

- HpkeConfigList Section 4.3.1: "application/dap-hpke-config-list"
- Report Section 4.3.2: "application/dap-report"
- AggregateInitializeReq Section 4.5: "application/dap-aggregate-initialize-req"
- AggregateInitializeResp Section 4.5: "application/dap-aggregate-initialize-resp"
- AggregateContinueReq Section 4.5: "application/dap-aggregate-continue-req"
- AggregateContinueResp Section 4.5: "application/dap-aggregate-continue-resp"
- AggregateShareReq Section 4.5: "application/dap-aggregate-share-req"
- AggregateShareResp Section 4.5: "application/dap-aggregate-share-resp"
- CollectReq Section 4.5: "application/dap-collect-req"
- CollectResp Section 4.5: "application/dap-collect-resp"

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

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

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

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

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

Type name:
application

Subtype name: dap-hpke-config-list

Required parameters: N/A

Optional parameters: None

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

Security considerations: see Section 4.2

Interoperability considerations: N/A

Published specification: this specification

Applications that use this media type: N/A

Fragment identifier considerations: N/A

Additional information:

Magic number(s): N/A

Deprecated alias names for this type: N/A

File extension(s): N/A

Macintosh file type code(s): N/A

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

Intended usage: COMMON

Restrictions on usage: N/A

Author: see Authors' Addresses section

Change controller: IESG

8.1.2. "application/dap-report" media type

Type name: application

Subtype name: dap-report

Required parameters: N/A

Optional parameters: None

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

Security considerations: see Section 4.3.2

Interoperability considerations: N/A

Published specification: this specification

Applications that use this media type: N/A

Fragment identifier considerations: N/A

Additional information:

Magic number(s): N/A

Deprecated alias names for this type: N/A

File extension(s): N/A

Macintosh file type code(s): N/A

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

Intended usage: COMMON

Restrictions on usage: N/A

Author: see Authors' Addresses section

Change controller: IESG

8.1.3. "application/dap-aggregate-initialize-req" media type

Type name: application

Subtype name: dap-aggregate-initialize-req

Required parameters: N/A

Optional parameters: None

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

Security considerations: see Section 4.5

Interoperability considerations: N/A

Published specification: this specification

Applications that use this media type:
N/A

Fragment identifier considerations: N/A

Additional information:

Magic number(s): N/A

Deprecated alias names for this type: N/A

File extension(s): N/A

Macintosh file type code(s): N/A

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

Intended usage: COMMON

Restrictions on usage: N/A

Author: see Authors' Addresses section

Change controller: IESG

8.1.4. "application/dap-aggregate-initialize-resp" media type

Type name: application

Subtype name: dap-aggregate-initialize-resp

Required parameters: N/A

Optional parameters: None

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

Security considerations: see Section 4.5

Interoperability considerations: N/A

Published specification: this specification

Applications that use this media type: N/A

Fragment identifier considerations: N/A

Additional information:

Magic number(s): N/A

Deprecated alias names for this type:
N/A

File extension(s): N/A

Macintosh file type code(s): N/A

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

Intended usage: COMMON

Restrictions on usage: N/A

Author: see Authors' Addresses section

Change controller: IESG

8.1.5. "application/dap-aggregate-continue-req" media type

Type name: application

Subtype name: dap-aggregate-continue-req

Required parameters: N/A

Optional parameters: None

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

Security considerations: see Section 4.5

Interoperability considerations: N/A

Published specification: this specification

Applications that use this media type: N/A

Fragment identifier considerations: N/A

Additional information:

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

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

Intended usage: COMMON

Restrictions on usage: N/A

Author: see Authors' Addresses section

Change controller: IESG

8.1.6. "application/dap-aggregate-continue-resp" media type

Type name: application

Subtype name: dap-aggregate-continue-resp

Required parameters: N/A

Optional parameters:
None

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

Security considerations: see Section 4.5

Interoperability considerations: N/A

Published specification: this specification

Applications that use this media type: N/A

Fragment identifier considerations: N/A

Additional information:

Magic number(s): N/A

Deprecated alias names for this type: N/A

File extension(s): N/A

Macintosh file type code(s): N/A

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

Intended usage: COMMON

Restrictions on usage: N/A

Author: see Authors' Addresses section

Change controller: IESG

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

Type name: application

Subtype name: dap-aggregate-share-req

Required parameters: N/A

Optional parameters: None

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

Security considerations: see Section 4.5

Interoperability considerations: N/A

Published specification:
this specification

Applications that use this media type: N/A

Fragment identifier considerations: N/A

Additional information:

Magic number(s): N/A

Deprecated alias names for this type: N/A

File extension(s): N/A

Macintosh file type code(s): N/A

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

Intended usage: COMMON

Restrictions on usage: N/A

Author: see Authors' Addresses section

Change controller: IESG

8.1.8. "application/dap-aggregate-share-resp" media type

Type name: application

Subtype name: dap-aggregate-share-resp

Required parameters: N/A

Optional parameters: None

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

Security considerations: see Section 4.5

Interoperability considerations: N/A

Published specification: this specification

Applications that use this media type: N/A

Fragment identifier considerations: N/A

Additional information:

Magic number(s):
N/A

Deprecated alias names for this type: N/A

File extension(s): N/A

Macintosh file type code(s): N/A

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

Intended usage: COMMON

Restrictions on usage: N/A

Author: see Authors' Addresses section

Change controller: IESG

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

Type name: application

Subtype name: dap-collect-req

Required parameters: N/A

Optional parameters: None

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

Security considerations: see Section 4.5

Interoperability considerations: N/A

Published specification: this specification

Applications that use this media type: N/A

Fragment identifier considerations: N/A

Additional information:

Magic number(s): N/A

Deprecated alias names for this type: N/A

File extension(s): N/A

Macintosh file type code(s): N/A

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

Intended usage: COMMON

Restrictions on usage: N/A

Author: see Authors' Addresses section

Change controller: IESG

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

Type name: application

Subtype name: dap-collect-req

Required parameters: N/A

Optional parameters: None

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

Security considerations: see Section 4.5

Interoperability considerations: N/A

Published specification: this specification

Applications that use this media type: N/A

Fragment identifier considerations: N/A

Additional information:

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

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

Intended usage: COMMON

Restrictions on usage: N/A

Author:
see Authors' Addresses section

Change controller: IESG

8.2. Query Types Registry

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

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

8.3. Upload Extension Registry

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

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

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

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

Registry name: dap

Specification: [[THIS DOCUMENT]]

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

Index value: No transformation needed.

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

9. Acknowledgments

The text in Section 3 is based extensively on [RFC8555]

10. References

10.1. 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>.

[I-D.thomson-http-oblivious]
Thomson, M. and C. A. Wood, "Oblivious HTTP", Work in Progress, Internet-Draft, draft-thomson-http-oblivious-02, 24 August 2021, <https://datatracker.ietf.org/doc/html/draft-thomson-http-oblivious-02>.

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

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

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

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

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

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

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

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

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

[RFC9111] Fielding, R., Ed., Nottingham, M., Ed., and J. Reschke, Ed., "HTTP Caching", STD 98, RFC 9111, DOI 10.17487/RFC9111, June 2022, <https://www.rfc-editor.org/rfc/rfc9111>.

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

[VDAF] Barnes, R., Patton, C., and P. Schoppmann, "Verifiable Distributed Aggregation Functions", Work in Progress, Internet-Draft, draft-irtf-cfrg-vdaf-03, 24 August 2022, <https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-vdaf-03>.

10.2. Informative References

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

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

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

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

[RFC8555] Barnes, R., Hoffman-Andrews, J., McCarney, D., and J. Kasten, "Automatic Certificate Management Environment (ACME)", RFC 8555, DOI 10.17487/RFC8555, March 2019, <https://www.rfc-editor.org/rfc/rfc8555>.

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

Authors' Addresses

Tim Geoghegan
ISRG

Email: timgeog+ietf@gmail.com

Christopher Patton
Cloudflare

Email: chrispatton+ietf@gmail.com

Eric Rescorla
Mozilla

Email: ekr@rtfm.com

Christopher A. Wood
Cloudflare

Email: caw@heapingbits.net