

Network Working Group  
Internet-Draft  
Intended status: Standards Track  
Expires: 4 November 2022

T. Geoghegan  
ISRG  
C. Patton  
Cloudflare  
E. Rescorla  
Mozilla  
C. A. Wood  
Cloudflare  
3 May 2022

Distributed Aggregation Protocol for Privacy Preserving Measurement  
draft-ietf-ppm-dap-00

## 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.

## Discussion Venues

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

Discussion of this document takes place on the mailing list (), which is archived at .

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

## 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-Draft

DAP-PPM

May 2022

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 4 November 2022.

## Copyright Notice

Copyright (c) 2022 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to [BCP 78](#) and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Revised BSD License text as described in Section 4.e of the [Trust Legal Provisions](#) and are provided without warranty as described in the Revised BSD License.

## Table of Contents

<a href="#">1.</a>	Introduction . . . . .	<a href="#">4</a>
<a href="#">1.1.</a>	DISCLAIMER . . . . .	<a href="#">4</a>
<a href="#">1.2.</a>	Conventions and Definitions . . . . .	<a href="#">4</a>
<a href="#">2.</a>	Overview . . . . .	<a href="#">5</a>
<a href="#">2.1.</a>	System Architecture . . . . .	<a href="#">6</a>
<a href="#">2.2.</a>	Validating Inputs . . . . .	<a href="#">9</a>
<a href="#">3.</a>	Message Transport . . . . .	<a href="#">9</a>
<a href="#">3.1.</a>	Errors . . . . .	<a href="#">10</a>
<a href="#">4.</a>	Protocol Definition . . . . .	<a href="#">11</a>
<a href="#">4.1.</a>	Task Configuration . . . . .	<a href="#">12</a>
<a href="#">4.2.</a>	Uploading Reports . . . . .	<a href="#">13</a>
<a href="#">4.2.1.</a>	Key Configuration Request . . . . .	<a href="#">14</a>
<a href="#">4.2.2.</a>	Upload Request . . . . .	<a href="#">15</a>
<a href="#">4.2.3.</a>	Upload Extensions . . . . .	<a href="#">16</a>
<a href="#">4.2.4.</a>	Leader State . . . . .	<a href="#">17</a>
<a href="#">4.3.</a>	Verifying and Aggregating Reports . . . . .	<a href="#">17</a>
<a href="#">4.3.1.</a>	Aggregate Initialization . . . . .	<a href="#">19</a>
<a href="#">4.3.2.</a>	Aggregate Continuation . . . . .	<a href="#">24</a>
<a href="#">4.4.</a>	Collecting Results . . . . .	<a href="#">26</a>
<a href="#">4.4.1.</a>	Collection Initialization . . . . .	<a href="#">27</a>

<a href="#">4.4.2.</a>	Collection Aggregation . . . . .	<a href="#">28</a>
<a href="#">4.4.3.</a>	Collection Finalization . . . . .	<a href="#">30</a>
<a href="#">4.4.4.</a>	Aggregate Share Encryption . . . . .	<a href="#">31</a>
<a href="#">4.4.5.</a>	Validating Batch Parameters . . . . .	<a href="#">31</a>
<a href="#">4.4.6.</a>	Anti-replay . . . . .	<a href="#">32</a>

<a href="#">5.</a>	Operational Considerations . . . . .	<a href="#">32</a>
<a href="#">5.1.</a>	Protocol participant capabilities . . . . .	<a href="#">33</a>
<a href="#">5.1.1.</a>	Client capabilities . . . . .	<a href="#">33</a>
<a href="#">5.1.2.</a>	Aggregator capabilities . . . . .	<a href="#">33</a>
<a href="#">5.1.3.</a>	Collector capabilities . . . . .	<a href="#">34</a>
<a href="#">5.2.</a>	Data resolution limitations . . . . .	<a href="#">34</a>
<a href="#">5.3.</a>	Aggregation utility and soft batch deadlines . . . . .	<a href="#">35</a>
<a href="#">5.4.</a>	Protocol-specific optimizations . . . . .	<a href="#">35</a>
<a href="#">5.4.1.</a>	Reducing storage requirements . . . . .	<a href="#">35</a>
<a href="#">6.</a>	Security Considerations . . . . .	<a href="#">36</a>
<a href="#">6.1.</a>	Threat model . . . . .	<a href="#">36</a>
<a href="#">6.1.1.</a>	Client/user . . . . .	<a href="#">37</a>
<a href="#">6.1.2.</a>	Aggregator . . . . .	<a href="#">37</a>
<a href="#">6.1.3.</a>	Leader . . . . .	<a href="#">39</a>
<a href="#">6.1.4.</a>	Collector . . . . .	<a href="#">40</a>
<a href="#">6.1.5.</a>	Aggregator collusion . . . . .	<a href="#">40</a>
<a href="#">6.1.6.</a>	Attacker on the network . . . . .	<a href="#">40</a>
<a href="#">6.2.</a>	Client authentication or attestation . . . . .	<a href="#">42</a>
<a href="#">6.3.</a>	Anonymizing proxies . . . . .	<a href="#">42</a>
<a href="#">6.4.</a>	Batch parameters . . . . .	<a href="#">42</a>
<a href="#">6.5.</a>	Differential privacy . . . . .	<a href="#">42</a>
<a href="#">6.6.</a>	Robustness in the presence of malicious servers . . . . .	<a href="#">43</a>
<a href="#">6.7.</a>	Infrastructure diversity . . . . .	<a href="#">43</a>
<a href="#">6.8.</a>	System requirements . . . . .	<a href="#">43</a>
<a href="#">6.8.1.</a>	Data types . . . . .	<a href="#">43</a>
<a href="#">7.</a>	IANA Considerations . . . . .	<a href="#">43</a>
<a href="#">7.1.</a>	Protocol Message Media Types . . . . .	<a href="#">43</a>
<a href="#">7.1.1.</a>	"application/ppm-hpke-config" media type . . . . .	<a href="#">44</a>
<a href="#">7.1.2.</a>	"message/ppm-report" media type . . . . .	<a href="#">45</a>
<a href="#">7.1.3.</a>	"message/ppm-aggregate-continue-req" media type . . . . .	<a href="#">46</a>
<a href="#">7.1.4.</a>	"message/ppm-aggregate-init-resp" media type . . . . .	<a href="#">47</a>
<a href="#">7.1.5.</a>	"message/ppm-aggregate-continue-req" media type . . . . .	<a href="#">48</a>
<a href="#">7.1.6.</a>	"message/ppm-aggregate-continue-resp" media type . . . . .	<a href="#">48</a>
<a href="#">7.1.7.</a>	"message/ppm-aggregate-share-req" media type . . . . .	<a href="#">49</a>
<a href="#">7.1.8.</a>	"message/ppm-aggregate-share-resp" media type . . . . .	<a href="#">50</a>
<a href="#">7.1.9.</a>	"message/ppm-collect-req" media type . . . . .	<a href="#">51</a>

<a href="#">7.1.10.</a> "message/ppm-collect-req" media type . . . . .	<a href="#">52</a>
<a href="#">7.2.</a> Upload Extension Registry . . . . .	<a href="#">53</a>
<a href="#">7.3.</a> URN Sub-namespace for PPM (urn:ietf:params:ppm) . . . . .	<a href="#">53</a>
<a href="#">8.</a> Acknowledgements . . . . .	<a href="#">53</a>
<a href="#">9.</a> References . . . . .	<a href="#">53</a>
<a href="#">9.1.</a> Normative References . . . . .	<a href="#">53</a>
<a href="#">9.2.</a> Informative References . . . . .	<a href="#">54</a>
Authors' Addresses . . . . .	<a href="#">55</a>

## [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.](#) DISCLAIMER

This document is a work in progress. We have not yet settled on the design of the protocol framework or the set of features we intend to support.

### [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:

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.

**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.

**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 PPM 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.

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

**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.

**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 PPM 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. We call the servers the `_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  $x_1, \dots, x_n$  held by  $n$  users, the goal of a `_privacy preserving measurement (PPM) protocol_` is to compute  $y = F(p, x_1, \dots, x_n)$  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

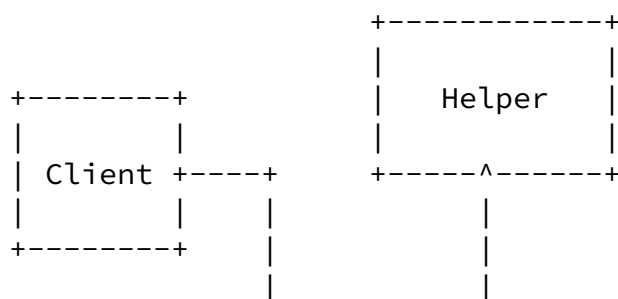
preserving manner.

This protocol is designed to work with schemes that use secret sharing. Rather than send 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's 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.



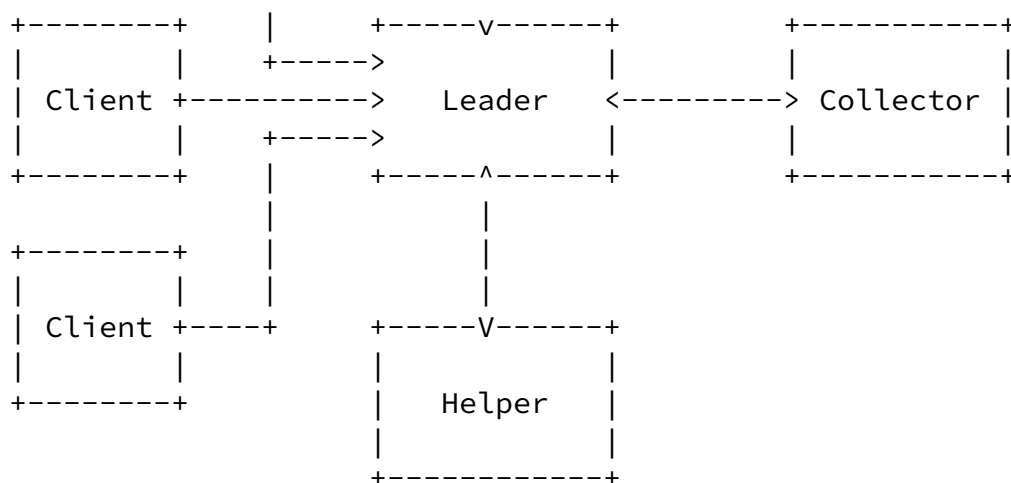


Figure 1: System Architecture

[[OPEN ISSUE: This shows two helpers, but the document only allows one for now. <https://github.com/ietf-wg-ppm/ppm-specification/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 PPM system. 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 PPM 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.) and an optional aggregation parameter.
- \* 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. 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 PPM, 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, if we want 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, if we wanted 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 PPM does not attempt to address it; it merely ensures that the data is within acceptable limits, so the client could not report  $10^6$ s or -20s.

## [3.](#) Message Transport

Communications between PPM entities are carried over HTTPS [[RFC2818](#)]. HTTPS provides server authentication and confidentiality. In addition, report shares are encrypted directly to the aggregators using HPKE [[I-D.irtf-cfrg-hpke](#)].

### [3.1.](#) Errors

Errors can be reported in PPM both at the HTTP layer and within challenge objects as defined in [Section 7](#). PPM 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 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 PPM URN namespace "urn:ietf:params:ppm: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.
outdatedConfig	The message was generated using an outdated configuration.
batchInvalid	A collect or aggregate-share request was made with invalid batch parameters.
batchMismatch	Aggregators disagree on the report shares that were aggregated in a batch.

Table 1

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

[Section 7.3](#)). Clients SHOULD display the "detail" field of all errors. The "instance" value MUST be the endpoint to which the request was targeted. The problem document MUST also include a "taskid" member which contains the associated PPM task ID, encoded with base64 using the standard alphabet [[RFC4648](#)] (this value is always known, see [Section 4.1](#)).

In the remainder of this document, we use the tokens in the table above 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:error:unrecognizedMessage".

This document uses the verbs "abort" and "alert with [some error message]" to describe how protocol participants react to various error conditions.

#### [4.](#) Protocol Definition

PPM has three major interactions which need to be defined:

- \* Uploading reports from the client to the aggregators, specified in [Section 4.2](#)
- \* Computing the results of a given measurement, specified in [Section 4.3](#)
- \* Collecting aggregated results, specified in [Section 4.4](#)

We start with some basic type definitions used in other messages.

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

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

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

```
/* An interval of time of length duration, where start is included and (start +
duration) is excluded. */
```

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

```
/* A nonce used to uniquely identify a report in the context of a PPM task. It
includes the time at which the report was generated and a random, 64-bit
integer. */
```

```
struct {
    Time time;
    uint64 rand;
} Nonce;
```

```
/* The various roles in the PPM protocol. */
```

```
enum {
    collector(0),
```

```

    client(1),
    leader(2),
    helper(3),
} 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 context
    opaque payload<1..2^16-1>; // ciphertext
} HpkeCiphertext;

```

#### [4.1](#). 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 4.2](#)) MUST be the same as the order in which aggregators appear in this list.
- \* max\_batch\_lifetime: The maximum number of times a batch of reports may be used in collect requests.
- \* min\_batch\_size: The minimum number of reports that appear in a batch.

- \* `min_batch_duration`: The minimum time difference between the oldest and newest report in a batch. This defines the boundaries with which the batch interval of each collect request must be aligned. (See [Section 4.4.5](#).)
- \* 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 4.2.1](#)).
- \* `vdaf_verify_param`: The aggregator's VDAF verification parameter output by the setup algorithm computed jointly by the aggregators before the start of the PPM protocol [[VDAF](#)). [OPEN ISSUE: This is yet to be specified. See issue#161.]

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

## [4.2](#). Uploading Reports

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

### [4.2.1](#). Key Configuration Request

Before the client can upload its report to the leader, it must know the public key of each of the aggregators. These are retrieved from each aggregator by sending a request to `[aggregator]/key_config`, where `[aggregator]` is the aggregator's endpoint URL, obtained from the task parameters. The aggregator responds to well-formed requests with status 200 and an `HpkeConfig` value:

```
struct {
    HpkeConfigId id;
```

```
    HpkeKemId kem_id;
    HpkeKdfId kdf_id;
    HpkeAeadKdfId aead_id;
    HpkePublicKey public_key;
} HpkeConfig;
```

```
opaque HpkePublicKey<1..2^16-1>;
uint16 HpkeAeadId; // Defined in I-D.irtf-cfrg-hpke
uint16 HpkeKemId;  // Defined in I-D.irtf-cfrg-hpke
uint16 HpkeKdfId;  // Defined in I-D.irtf-cfrg-hpke
```

[OPEN ISSUE: Decide whether to expand the width of the id, or support multiple cipher suites (a la OHTTP/ECH).]

The client MUST abort if any of the following happen for any key\_config request:

- \* the client and aggregator failed to establish a secure, aggregator-authenticated channel;
- \* the GET request failed or didn't return a valid key config; or
- \* the key config specifies a KEM, KDF, or AEAD algorithm the client doesn't 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 [[RFC7234](#)] semantics for key configurations.

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



#### [4.2.2.](#) 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 {  
    TaskID task_id;  
    Nonce nonce;  
    Extension extensions<0.. $2^{16}-1$ >;  
    HpkeCiphertext encrypted_input_shares<1.. $2^{16}-1$ >;  
} Report;
```

This message is called the client's `_report_`. It contains the following fields:

- \* `task_id` is the task ID of the task for which the report is intended.
- \* `nonce` is the report nonce generated by the client. This field is used by the aggregators to ensure the report appears in at most one batch. (See [Section 4.4.6.](#))
- \* `extensions` is a list of extensions to be included in the Upload flow; see [Section 4.2.3.](#)
- \* `encrypted_input_shares` contains the encrypted input shares of each of the aggregators. The order in which the encrypted input shares appear MUST match the order of the task's `aggregator_endpoints` (i.e., the first share should be the leader's, the second share should be for the first helper, and so on).

To generate the report, the client begins by sharding its measurement into a sequence of input shares as specified by the VDAF in use. To encrypt an input share, the client first generates an HPKE [\[I-D.irtf-cfrg-hpke\]](#) context for the aggregator by running

```
enc, context = SetupBaseS(pk, Report.task_id || "ppm-00 input share" ||  
                           0x01 || server_role)
```

where `pk` is the aggregator's public key and `server_role` is the Role of the intended recipient (0x02 for the leader and 0x03 for the helper). In general, the info string for computing the HPKE context

is suffixed by two bytes, the first of which identifies the role of the sender and the second of which identifies the role of the intended recipient.

enc is the HPKE encapsulated key and context is the HPKE context used by the client for encryption. The payload is encrypted as

```
payload = context.Seal(nonce || extensions, input_share)
```

where input\_share is the aggregator's input share and nonce and extensions are the corresponding fields of Report. Clients MUST NOT use the same enc for multiple reports.

The leader responds to well-formed requests to [leader]/upload with status 200 and an empty body. Malformed requests are handled as described in [Section 3.1](#). Clients SHOULD NOT upload the same measurement value in more than one report if the leader responds with status 200 and an empty body.

The leader responds to requests whose leader encrypted input share uses an out-of-date HpkeConfig.id value, indicated by HpkeCiphertext.config\_id, with status 400 and an error of type 'outdatedConfig'. Clients SHOULD invalidate any cached aggregator HpkeConfig and retry with a freshly generated Report. If this retried report does not succeed, clients MUST abort and discontinue retrying.

The leader MUST ignore any report whose nonce contains a timestamp that falls in a batch interval for which it has received at least one collect request from the collector. (See [Section 4.4](#).) Otherwise, comparing the aggregate result to the previous aggregate result may result in a privacy violation. (Note that the helpers enforce this as well; see [Section 4.4](#).) In addition, the leader SHOULD abort the upload protocol and alert the client with error "staleReport".

#### [4.2.3](#). Upload Extensions

Each Report carries a list of extensions that clients may use to convey additional, authenticated information in the report. [OPEN ISSUE: The extensions aren't authenticated. It's probably a good idea to be a bit more clear about how we envision extensions being used. Right now this includes client attestation for defeating Sybil attacks. See issue#89.] Each extension is a tag-length encoded value of the following form:

Internet-Draft

DAP-PPM

May 2022

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

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

"extension\_type" indicates the type of extension, and  
"extension\_data" contains information specific to the extension.

#### [4.2.4.](#) Leader State

The leader MUST 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.

#### [4.3.](#) 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. To aggregate a set of reports, the leader sends a message to each helper containing those report shares. 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 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.

In order to allow the helpers to retain minimal state, the helper can attach a state parameter to its response, with the leader returning the state value in the next request, thus offloading the state to the leader. This state value **MUST** be cryptographically protected as described in [Section 4.3.1](#).

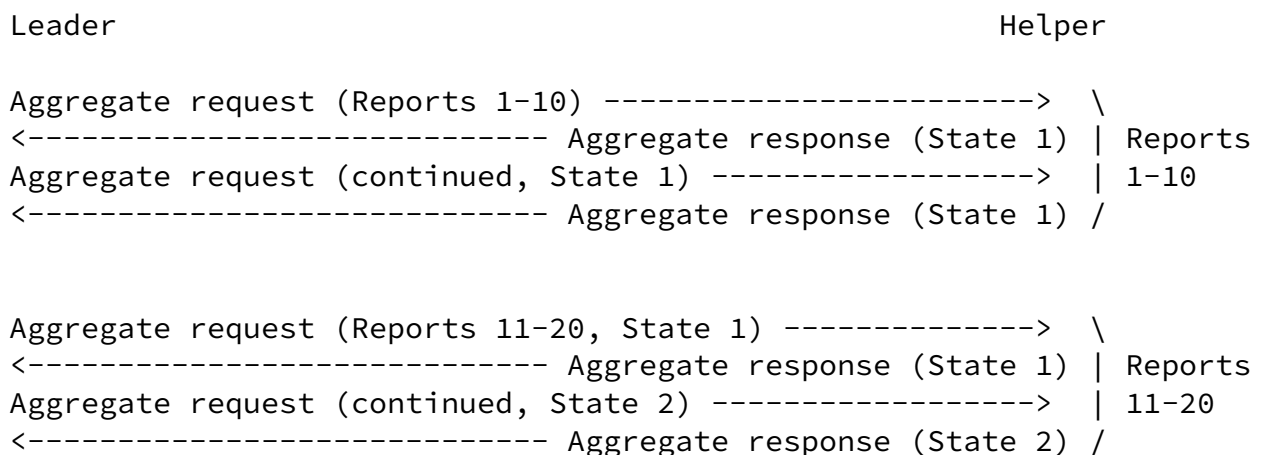


Figure 2: Aggregation Flow (batch size=20). Multiple aggregation flows can be executed at the same time.

[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.

#### [4.3.1.](#) Aggregate Initialization

The leader begins aggregation by choosing a set of candidate reports that pertain to the same PPM task. 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),  
} ReportShareError;
```

The leader and helper initialization behavior is detailed below.

#### [4.3.1.1.](#) Leader Initialization

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

1. Decrypt the input share for each report share as described in [Section 4.3.1.3.](#)
2. Check that the resulting input share is valid as described in [Section 4.3.1.4.](#)
3. Initialize VDAF preparation as described in [Section 4.3.1.5.](#)

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 AggregateInitReq message for each helper to initialize the preparation of this candidate set. The AggregateInitReq message is structured as follows:

```
struct {  
    Nonce nonce;  
    Extension extensions<0..2^16-1>;  
    HpkeCiphertext encrypted_input_share;  
} ReportShare;
```

```
struct {  
    TaskID task_id;  
    opaque agg_param<0..2^16-1>;  
    opaque helper_state<0..2^16>;  
    ReportShare report_shares<1..2^16-1>;  
} AggregateInitReq;
```

[[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. ]]

The nonce and extensions fields of each ReportShare match that in the

Report uploaded by the client. The `encrypted_input_share` field is the `HpkcCiphertext` whose index in `Report.encrypted_input_shares` is equal to the index of the aggregator in the task's `aggregator_endpoints` to which the `AggregateInitReq` is being sent. The `agg_param` field is an opaque, VDAF-specific aggregation parameter. The `helper_state` parameter contains the helper's state. This is an optional parameter of an aggregate request that the helper can use to carry state across requests and across aggregate flows.

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

#### [4.3.1.2.](#) Helper Initialization

Each helper begins their portion of the aggregate initialization phase with the set of candidate report shares obtained in an `AggregateInitReq` 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 that the nonces in `AggregateInitReq.report_shares` are all distinct. If two `ReportShare` values have the same nonce, then the helper MUST abort with error "unrecognizedMessage". If this check succeeds, the helper then attempts to recover each input share in `AggregateInitReq.report_shares` as follows:

1. Decrypt the input share for each report share as described in [Section 4.3.1.3](#).
2. Check that the resulting input share is valid as described in [Section 4.3.1.4](#).
3. Initialize VDAF preparation and initial outputs as described in [Section 4.3.1.5](#).

[[OPEN ISSUE: consider moving the helper nonce check into #input-share-batch-validation]]

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

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

struct {
    Nonce nonce;
    PrepareStepResult prepare_step_result;
    select (PrepareStep.prepare_step_result) {
        case continued: opaque prep_msg<0..2^16-1>; // VDAF preparation message
        case finished: Empty;
        case failed: ReportShareError;
    }
} PrepareStep;

struct {
    opaque helper_state<0..2^16>;
    PrepareStep prepare_steps<1..2^16-1>;
} AggregateInitResp;
```

The `helper_state` parameter contains the helper's initial state. This is an optional parameter of an aggregate request that the helper can use to carry state across requests. At least part of the state will usually need to be encrypted in order to protect user privacy. However, the details of precisely how the state is encrypted and the information that it carries is up to the helper implementation.

[[OPEN ISSUE: we may end up removing `helper_state`. See #185]]

The rest of the message is a sequence of `PrepareStep` values, the order of which matches that of the `ReportShare` values in `AggregateInitReq.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 200 OK whose body is the `AggregateInitResp` and media type is `"message/ppm-aggregate-init-resp"`.

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

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

#### [4.3.1.3](#). Input Share Decryption

Each report share has a corresponding task ID, nonce, list of extensions, and encrypted input share. Let `task_id`, `nonce`, `extensions`, and `encrypted_input_share` denote these values, respectively. Given these values, an aggregator decrypts the input share as follows. First, the aggregator looks up the HPKE config and corresponding secret key indicated by `encrypted_input_share.config_id`. If not found, then it marks the report share as invalid with the error `hpke-unknown-config-id`. Otherwise, it decrypts the payload with the following procedure:

```
context = SetupBaseR(encrypted_input_share.enc, sk, task_id ||
                    "ppm-00 input share" || 0x01 || server_role)

input_share = context.Open(nonce || extensions,
                          encrypted_input_share.payload)
```

where `sk` is the HPKE secret key, `task_id` is the task ID, `nonce` and `extensions` are the nonce and extensions of the report share respectively, and `server_role` is `0x02` if the aggregator is the leader and `0x03` otherwise. If decryption fails, the aggregator marks the report share as invalid with the error `hpke-decrypt-error`. Otherwise, it outputs the resulting `input_share`.

#### [4.3.1.4.](#) 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.

The validation checks are as follows.

1. 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 is marked as invalid with the error `batch-collected`. This prevents additional reports from being aggregated after its batch has already been collected.
2. Check if the report has already been aggregated. If this check fails, the input share is 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. An aggregator may also choose to mark an input share as invalid with the error `report-dropped` under the conditions prescribed in [Section 4.4.6](#).

If both checks succeed, the input share is not marked as invalid.

#### [4.3.1.5.](#) 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 4.4](#).
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 nonce as follows:

```
prep_state = VDAF.prep_init(vdaf_verify_param, agg_param, nonce, input_share)
out = VDAF.prep_next(prepare_state, None)
```

Internet-Draft

DAP-PPM

May 2022

`vdaf_verify_param` is the public VDAF parameter, and `agg_param` is the opaque aggregation parameter. 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)`.

#### [4.3.2.](#) 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 parameters. Each round trip is initiated by the leader.

##### [4.3.2.1.](#) 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 marks the report as failed and removes it from the candidate report set and does not process it further. If the type is `"finished"`, then the leader aborts with `"unrecognizedMessage"`. [[OPEN ISSUE: This behavior is not specified.]] 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_outbound])
out = VDAF.prep_next(prepare_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 4.4](#). 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 {  
    opaque helper_state<0..2^16>;  
    PrepareStep prepare_shares<1..2^16-1>;  
} 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 `"message/ppm-aggregate-continue-req"`.

#### [4.3.2.2](#). Helper Continuation

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 4.4](#).
3. An updated VDAF state and preparation message, denoted (`prep_state`, `prep_msg`).

To carry out this step, for each `PrepareStep` in `AggregateContinueReq.prepare_shares` 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 4.4](#).

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

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

where inbound is the previous VDAF prepare message sent by the leader and prepare\_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\_shares is then sent to the leader in an AggregateContinueResp message, structured as follows:

```
struct {  
    opaque helper_state<0..2^16>;  
    PrepareStep prepare_shares<1..2^16-1>;  
} AggregateContinueResp;
```

The order of AggregateContinueResp.prepare\_shares MUST match that of the PrepareStep values in AggregateContinueReq.prepare\_shares. The helper's response to the leader is an HTTP 200 OK whose body is the AggregateContinueResp and media type is "message/ppm-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](#). Collecting Results

In this phase, the collector requests aggregate shares from each

aggregator and then locally combines them to yield a single, aggregate output. In particular, the collector asks the leader to collect and return the results for a given PPM task over a given time period. The aggregate shares are encrypted to the collector so that it can decrypt and combine them to yield the aggregate output. This entire process is composed of two interactions:

1. Collect request and response between the collector and leader, specified in [Section 4.4.1](#)
2. Aggregate share request and response between the leader and each aggregator, specified in [Section 4.4.2](#)

Once complete, the collector computes the final aggregate result as specified in [Section 4.4.3](#).

#### [4.4.1](#). 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;  
    Interval batch_interval;  
    opaque agg_param<0..2^16-1>; // VDAF aggregation parameter  
} CollectReq;
```

The named parameters are:

- \* task\_id, the PPM task ID.
- \* batch\_interval, the request's batch interval.

\* `agg_param`, an aggregation parameter for the VDAF being executed.

Depending on the VDAF scheme and how the leader is configured, the leader and helper may already have prepared all the reports falling within `batch_interval` and be ready to return the aggregate shares right away, but this cannot be guaranteed. In fact, for some VDAFs, it is not 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 4.4.5](#). 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 falling into `CollectReq.batch_interval` (or continues this process, depending on the VDAF) as described in [Section 4.3](#).

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 4.4.2](#). 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 200 OK and a body consisting of a `CollectResp`:

```
struct {  
    HpkeCiphertext encrypted_agg_shares<1..2^16-1>;  
} CollectResp;
```

The `encrypted_agg_shares` field is 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 3.1](#).

The leader MUST retain a collect job's results until the collector sends an HTTP DELETE request to the collect job URI, in which case the leader responds with HTTP status 204 No Content.

[OPEN ISSUE: Allow the leader to drop aggregate shares after some reasonable amount of time has passed, but it's not clear how to specify that. ACME doesn't bother to say anything at all about this when describing how subscribers should fetch certificates:  
<https://datatracker.ietf.org/doc/html/rfc8555#section-7.4.2>]

[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?]

#### [4.4.2](#). 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 window identified by the collect request. The checksum is computed by taking the SHA256 hash of each nonce 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 4.4](#)) except its own, the leader sends a POST request to `[aggregator]/aggregate_share` with the following message:



```

struct {
    TaskID task_id;
    Interval batch_interval;
    uint64 report_count;
    opaque checksum[32];
    opaque helper_state<0..2^16>;
} AggregateShareReq;

```

- \* task\_id is the task ID associated with the PPM parameters.
- \* batch\_interval is the batch interval of the request.
- \* report\_count is the number of reports included in the aggregation.
- \* checksum is the checksum computed over the set of client reports.
- \* helper\_state is the helper's state, which is carried across requests from the leader.

To handle the leader's request, the helper first ensures that the request meets the requirements for batch parameters following the procedure in [Section 4.4.5](#). If the batch parameters are invalid, then it MUST abort with error "batchInvalid".

Next, it computes a checksum based on its view of the output shares included in the batch window, and checks that the report\_count and checksum included in the request match its computed values. If not, then it MUST abort with error "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 4.4.5](#) so that the aggregator knows which aggregate share to update.

The helper then encrypts `agg_share` under the collector's HPKE public key as described in [Section 4.4.4](#), 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 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 4.4](#)).

After issuing an aggregate-share request for a given batch interval, it is an error for the leader to issue any more aggregate or aggregate-init requests for additional reports in the batch interval. These reports will be rejected by helpers as described [Section 4.3.1](#).

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 4.4.4](#).

#### [4.4.3](#). 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 4.4.4](#). 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, 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)
```

Internet-Draft

DAP-PPM

May 2022

#### [4.4.4.](#) Aggregate Share Encryption

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

```
enc, context = SetupBaseS(pk, AggregateShareReq.task_id ||
                          "ppm-00 aggregate share" || server_role || 0x00)
encrypted_agg_share = context.Seal(AggregateShareReq.batch_interval,
                                   agg_share)
```

where `pk` is the HPKE public key encoded by the collector's HPKE key, and `server_role` is `0x02` for the leader and `0x03` for a helper.

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

```
context = SetupBaseR(enc_share.enc, sk,
                     "ppm-00 aggregate share" ||
                     task_id || server_role || 0x00)
agg_share = context.Open(batch_interval, enc_share.payload)
```

where `sk` is the HPKE secret key, `task_id` is the task ID for a given collect request, and `server_role` is the role of the server that sent the aggregate share (`0x02` for the leader and `0x03` for the helper).

#### [4.4.5.](#) Validating Batch Parameters

Before an aggregator responds to a collect request or aggregate-share request, it must first check that the request does not violate the parameters associated with the PPM task. It does so as described here.

First the aggregator checks that the request's batch interval respects the boundaries defined by the PPM task's parameters. Namely, it checks that both `batch_interval.start` and `batch_interval.duration` are divisible by `min_batch_duration` and that `batch_interval.duration >= min_batch_duration`. Unless both these conditions are true, it aborts and alerts the peer with "invalid batch interval".

Next, the aggregator checks that the request respects the generic privacy parameters of the PPM task. Let  $X$  denote the set of reports for which the aggregator has recovered a valid output share and which fall in the batch interval of the request.

- \* If  $\text{len}(X) < \text{min\_batch\_size}$ , then the aggregator aborts and alerts the peer with "insufficient batch size".
- \* The aggregator keeps track of the number of times each report was added to the batch of an `AggregateShareReq`. If any report in  $X$  was added to at least `max_batch_lifetime` previous batches, then the helper aborts and alerts the peer with "request exceeds the batch's privacy budget".

#### [4.4.6](#). Anti-replay

Using a client-provided report multiple times within a single batch, or using the same report in multiple batches, may allow a server to learn information about the client's measurement, violating the privacy goal of PPM. To prevent such replay attacks, this specification requires the aggregators to detect and filter out replayed reports.

To detect replay attacks, each aggregator keeps track of the set of nonces pertaining to reports that were previously aggregated for a given task. If the leader receives a report from a client whose nonce is in this set, it simply ignores it. A helper who receives an encrypted input share whose nonce is in this set replies to the leader with an error as described in [Section 4.4](#).

[OPEN ISSUE: This has the potential to require aggregators to store nonce sets indefinitely. See issue#180.]

A malicious aggregator may attempt to force a replay by replacing the nonce generated by the client with a nonce its peer has not yet seen. To prevent this, clients incorporate the nonce into the AAD for HPKE encryption, ensuring that the output share is only recovered if the aggregator is given the correct nonce. (See [Section 4.2.2](#).)

Aggregators prevent the same report from being used in multiple batches (except as required by the protocol) by only responding to

valid collect requests, as described in [Section 4.4.5](#).

## [5.](#) Operational Considerations

PPM protocols have 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 collect 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 PPM 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 with optional support for state offloading. Some of this state may be used for replay attack mitigation. The replay mitigation strategy is described in [Section 4.4.6](#).

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 PPM task, leaders are required to:

- \* Implement and store state for the form of inter- and intra-batch replay mitigation in [Section 4.4.6](#); and
- \* Store helper state.

### [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 PPM 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 all valid reports for as long as collect requests can be made for them. In particular, the aggregators must store a batch as long as the batch has not been queried more than `max_batch_lifetime` times. However, it is not always necessary to store the reports themselves. For schemes like Prio in which the input-validation protocol is only run once per report, 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. (The helper may store its aggregate shares in its encrypted state, thereby offloading this state to the leader.) This is due to the requirement that the batch interval respect the boundaries defined by the PPM parameters. (See [Section 4.4.5.](#))

## [6.](#) Security Considerations

Prio assumes a powerful adversary with the ability to compromise an unbounded number of clients. In doing so, the adversary can provide malicious (yet truthful) inputs to the aggregation function. Prio also assumes that all but one server operates honestly, where a dishonest server does not execute the protocol faithfully as specified. The system also assumes that servers communicate over



secure and mutually authenticated channels. In practice, this can be done by TLS or some other form of application-layer authentication.

In the presence of this adversary, Prio provides two important properties for computing an aggregation function  $F$ :

1. Privacy. The aggregators and collector learn only the output of  $F$  computed over all client inputs, and nothing else.
2. Robustness. As long as the aggregators execute the input-validation protocol correctly, a malicious client can skew the output of  $F$  only by reporting false (untruthful) input. The output cannot be influenced in any other way.

There are several additional constraints that a Prio deployment must satisfy in order to achieve these goals:

1. Minimum batch size. The aggregation batch size has an obvious impact on privacy. (A batch size of one hides nothing of the input.)
2. Aggregation function choice. Some aggregation functions leak slightly more than the function output itself.

[TODO: discuss these in more detail.]

### [6.1.](#) Threat model

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.

#### [6.1.1.](#) Client/user

#### [6.1.1.1.](#) Assets

1. Unshared inputs. Clients are the only actor that can ever see the original inputs.
2. Unencrypted input shares.

#### [6.1.1.2.](#) 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.

#### [6.1.1.3.](#) Mitigations

1. The input validation protocol executed by the aggregators prevents either individual clients or coalitions of clients from compromising the robustness property.
2. If aggregator output satisfies differential privacy [Section 6.5](#), then all records not leaked by malicious clients are still protected.

### [6.1.2.](#) Aggregator

#### [6.1.2.1.](#) Assets

1. Unencrypted input shares.
2. Input share decryption keys.
3. Client identifying information.

4. Aggregate shares.
5. Aggregator identity.

#### [6.1.2.2.](#) 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 6.5](#)) if the presence or absence of a share for a given user is sensitive.

#### [6.1.2.3.](#) 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.

Internet-Draft

DAP-PPM

May 2022

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.]

### [6.1.3.](#) Leader

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

#### [6.1.3.1.](#) 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.

#### [6.1.3.2.](#) Mitigations

Geoghegan, et al.

Expires 4 November 2022

[Page 39]

---

Internet-Draft

DAP-PPM

May 2022

1. Aggregators enforce agreed upon minimum aggregation thresholds to prevent deanonymizing.
2. If aggregator output satisfies differential privacy [Section 6.5](#), then genuine records are protected regardless of the size of the anonymity set.

#### [6.1.4.](#) Collector

##### [6.1.4.1.](#) 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.

##### [6.1.4.2.](#) 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 6.5](#), then genuine records are protected regardless of the size of the anonymity set.

#### [6.1.5.](#) 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.

#### [6.1.6.](#) Attacker on the network

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

##### [6.1.6.1.](#) 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.

##### [6.1.6.2.](#) 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 soundness) *\*or\** a malicious subset of aggregators (attacking privacy). In particular, soundness 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 soundness. 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.]]

## [6.2.](#) Client authentication or attestation

[TODO: Solve issue#89]

## [6.3.](#) 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 PPM leader, without requiring any server participating in PPM to be aware of whatever client authentication or attestation scheme is in use.

## [6.4.](#) Batch parameters

An important parameter of a PPM 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 PPM task if the minimum batch size is too small. This document does not specify how to choose minimum batch sizes.

The PPM 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.

## [6.5.](#) Differential privacy

Optionally, PPM 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.]

## [6.6.](#) Robustness in the presence of malicious servers

Most PPM 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 (<https://github.com/ietf-wg-ppm/ppm-specification/issues/22>) for discussion.

## [6.7.](#) 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.

## [6.8.](#) System requirements

### [6.8.1.](#) Data types

## [7.](#) IANA Considerations

### [7.1.](#) Protocol Message Media Types

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

- \* HpkeConfig [Section 4.1](#): "application/ppm-hpke-config"
- \* Report [Section 4.2.2](#): "message/ppm-report"
- \* AggregateInitReq [Section 4.4](#): "message/ppm-aggregate-init-req"
- \* AggregateInitResp [Section 4.4](#): "message/ppm-aggregate-init-resp"

- \* AggregateContinueReq [Section 4.4](#): "message/ppm-aggregate-continue-req"
- \* AggregateContinueResp [Section 4.4](#): "message/ppm-aggregate-

continue-resp"

- \* AggregateShareReq [Section 4.4](#): "message/ppm-aggregate-share-req"
- \* AggregateShareResp [Section 4.4](#): "message/ppm-aggregate-share-resp"
- \* CollectReq [Section 4.4](#): "message/ppm-collect-req"
- \* CollectResp [Section 4.4](#): "message/ppm-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.]

#### [7.1.1.1](#). "application/ppm-hpke-config" media type

Type name: application

Subtype name: ppm-hpke-config

Required parameters: N/A

Optional parameters: None

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

Security considerations: see [Section 4.1](#)

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

#### [7.1.2.](#) "message/ppm-report" media type

Type name: message

Subtype name: ppm-report

Required parameters: N/A

Optional parameters: None

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

Security considerations: see [Section 4.2.2](#)

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

Geoghegan, et al.

Expires 4 November 2022

[Page 45]

---

Internet-Draft

DAP-PPM

May 2022

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

#### [7.1.3.](#) "message/ppm-aggregate-continue-req" media type

Type name: message

Subtype name: ppm-aggregate-init-req

Required parameters: N/A

Optional parameters: None

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

Security considerations: see [Section 4.4](#)

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

Geoghegan, et al.

Expires 4 November 2022

[Page 46]

---

Internet-Draft

DAP-PPM

May 2022

Author: see Authors' Addresses section

Change controller: IESG

#### [7.1.4.](#) "message/ppm-aggregate-init-resp" media type

Type name: message

Subtype name: ppm-aggregate-init-resp

Required parameters: N/A

Optional parameters: None

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

Security considerations: see [Section 4.4](#)

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

Geoghegan, et al. Expires 4 November 2022 [Page 47]

---

Internet-Draft DAP-PPM May 2022

#### [7.1.5.](#) "message/ppm-aggregate-continue-req" media type

Type name: message

Subtype name: ppm-aggregate-continue-req

Required parameters: N/A

Optional parameters: None

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

Security considerations: see [Section 4.4](#)

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

#### [7.1.6.](#) "message/ppm-aggregate-continue-resp" media type

Type name: init

Subtype name: ppm-aggregate-continue-resp

Required parameters: N/A

Geoghegan, et al.

Expires 4 November 2022

[Page 48]

---

Internet-Draft

DAP-PPM

May 2022

Optional parameters: None

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

Security considerations: see [Section 4.4](#)

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

#### [7.1.7.](#) "message/ppm-aggregate-share-req" media type

Type name: message

Subtype name: ppm-aggregate-share-req

Required parameters: N/A

Optional parameters: None

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

Security considerations: see [Section 4.4](#)

Interoperability considerations: N/A

Geoghegan, et al.

Expires 4 November 2022

[Page 49]

---

Internet-Draft

DAP-PPM

May 2022

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

[7.1.8.](#) "message/ppm-aggregate-share-resp" media type

Type name: message

Subtype name: ppm-aggregate-share-resp

Required parameters: N/A

Optional parameters: None

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

Security considerations: see [Section 4.4](#)

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

#### [7.1.9.](#) "message/ppm-collect-req" media type

Type name: message

Subtype name: ppm-collect-req

Required parameters: N/A

Optional parameters: None

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

Security considerations: see [Section 4.4](#)

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

hous' Addresses section

Intended usage: COMMON

Restrictions on usage: N/A

Author: see Authors' Addresses section

Change controller: IESG

#### [7.1.10](#). "message/ppm-collect-req" media type

Type name: message

Subtype name: ppm-collect-req

Required parameters: N/A

Optional parameters: None

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

Security considerations: see [Section 4.4](#)

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

Internet-Draft

DAP-PPM

May 2022

Change controller: IESG

## [7.2.](#) Upload Extension Registry

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

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

## [7.3.](#) URN Sub-namespace for PPM (urn:ietf:params:ppm)

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: ppm

Specification: [[THIS DOCUMENT]]

Repository: <http://www.iana.org/assignments/ppm>

Index value: No transformation needed.

Initial contents: The types and descriptions in the table in [Section 3.1](#) above, with the Reference field set to point to this specification.

## [8.](#) Acknowledgements

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

## [9.](#) References

### [9.1.](#) Normative References

[I-D.irtf-cfrg-hpke]

Barnes, R. L., Bhargavan, K., Lipp, B., and C. A. Wood, "Hybrid Public Key Encryption", Work in Progress, Internet-Draft, [draft-irtf-cfrg-hpke-12](#), 2 September 2021, <<https://datatracker.ietf.org/doc/html/draft-irtf-cfrg->

Internet-Draft

DAP-PPM

May 2022

[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>>.

[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>>.

[RFC2818] Rescorla, E., "HTTP Over TLS", [RFC 2818](#), DOI 10.17487/RFC2818, May 2000, <<https://www.rfc-editor.org/rfc/rfc2818>>.

[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>>.

[RFC7234] Fielding, R., Ed., Nottingham, M., Ed., and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Caching", [RFC 7234](#), DOI 10.17487/RFC7234, June 2014, <<https://www.rfc-editor.org/rfc/rfc7234>>.

[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>>.

## [9.2](#). Informative References

Geoghegan, et al.

Expires 4 November 2022

[Page 54]

---

Internet-Draft

DAP-PPM

May 2022

- [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](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](https://privacytools.seas.harvard.edu/files/privacytools/files/complexityprivacy_1.pdf)>.

[VDAF] Barnes, R. L., Patton, C., and P. Schoppmann, "Verifiable Distributed Aggregation Functions", Work in Progress, Internet-Draft, [draft-irtf-cfrg-vdaf-00](https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-vdaf-00), 27 April 2022, <<https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-vdaf-00>>.

#### Authors' Addresses

Tim Geoghegan  
ISRG  
Email: [timgeog+ietf@gmail.com](mailto:timgeog+ietf@gmail.com)

Christopher Patton  
Cloudflare  
Email: [chrispatton+ietf@gmail.com](mailto:chrispatton+ietf@gmail.com)

Eric Rescorla  
Mozilla

Geoghegan, et al.

Expires 4 November 2022

[Page 55]

---

Internet-Draft

DAP-PPM

May 2022

Email: [ekr@rtfm.com](mailto:ekr@rtfm.com)

Christopher A. Wood  
Cloudflare  
Email: [caw@heapingbits.net](mailto:caw@heapingbits.net)

