openPDS Developer Documentation

Brian Sweatt (brian717@media.mit.edu)

Human Dynamics Group, MIT Media Lab

August 6, 2014

Contents

This document covers the core openPDS architecture, beginning with definition of terms, and proceeding to a description of the functional components necessary to build a cohesive personal data store solution. The functional components are organized into distinct sections: authentication service and Registry, individual decentralized Personal Data Stores, and REST service interfaces for both.

1 Nomenclature

Authorization Server A server issuing OAuth 2.0 access tokens to the Client after successfully authenticating the user and obtaining authorization.

Client In the OAuth context, an application or system making protected requests on behalf of the ser and with his or her authorization.

OAuth 2.0 An open protocol to allow secure API authorization in a simple and standard method from desktop and web applications. It enables users to grant third-party access to their web Resources without sharing their passwords.

Personal Data Store A protected resource owned and controlled by an individual to hold their personal data. The user controls access to use, modification, copying, derivative works, and redaction or deletion of the data they enter into the Personal Data Store, including data collected

from their smartphone via either passive sensor collection or surveys.

Registry Account management and database of registered users. This server stores only data that is necessary to authenticate a user (email and password hash) and locate their PDS, and has an internal as well as an external identifier for each user. The Registry authenticates login requests, as part of the OAuth authorization flow.

REST Service A type of web service that is stateless, cacheable, and provides a uniform interface to resources. Such services use URLs to uniquely identify resources and standard HTTP methods of GET, POST, PUT, and DELETE to specify the operation to perform on a resource.

Scope When an individual authorizes access to data on their personal data store, the access token also includes one or multiple named Scopes, each designating a type of data access that has been authorized.

Symbolic User ID Indirect reference to a registered users OAuth and Registry identifier key. Also referred to as a participants UUID.

User ID A participants OAuth and Registry identifier key. It is accessed only internally by openPDS and the Registry server. When Entities other than the Participant or System Entity need to reference it, they are given a Symbolic User ID.

2 High-Level System Diagram

Figure 1 provides a high-level overview of all stakeholders in a general personal data ecosystem. This includes third parties providing data and services, users in the ecosystem (shown as participants), and applications acting on behalf of users and services.

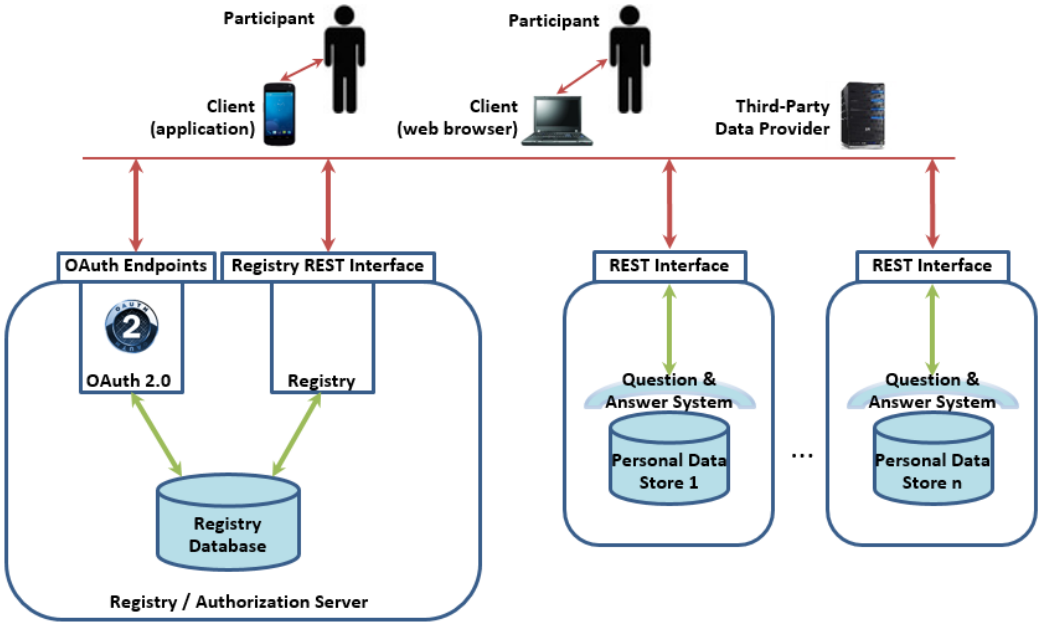

Figure 1: High-Level personal data ecosystem - a set of decentralized personal data stores constitute a trust framework with a centralized registry of users and authorization service.

Subsequent sections explain each of the blocks called out in figure 1, including Registry and authorization, personal data stores and their associated question and answer system, as well as clients and data providers built on top of the ecosystem. APIs are described to facilitate communication between physical servers in a personal data ecosystem, as well as how data is

organized in personal data stores, and the means by which one may define new applications or data connectors to run on personal data stores.

3 Registry and Authorization

An authorization server, supporting the OAuth 2.0 protocol, provides secure user authentication and authorization of access to personal data stores. The OAuth component is tightly coupled with a registry providing account management services. As part of these account management services, the registry includes a database of registered users, holding login credentials for authentication, and profile data.

Account management services include profile edits, password changes, and recovery procedures for forgotten passwords. Administrators have full access to these services, while other participants may have access only a restricted subset.

At the point of user registration, the registry creates a profile for the user, and a personal data store is lazily initialized for them. The user profile contains the following information:

Username used to login to the registry server and personal data ecosystem. Must be unique.

Email used for account verification purposes. Must be unique.

Password hash

First name

Last name

UUID anonymous symbolic identifier used to address this user across the personal data ecosystem. Is unique by definition.

PDS Location URL identifying where the user's distributed personal data store resides

Additionally, to facilitate authorization, the registry server also holds information about the groups a user belongs to, as well as the roles the user can take on within the system. After registration, authorization for collecting and utilizing data by an app on the user's behalf follows the flow described in figure 2. The registry server also holds all information pertaining to the flow described therein - including all access tokens and authorization codes for all users in the ecosystem. These users can revoke authorization tokens and, in turn, revoke the access they had provided to their personal data. Each request against any server in the personal data ecosystem must provide an Authorization header of the form: `Authorization: Bearer <token>`, where `<token>` is a placeholder for a valid authorization token.

In addition to the traditional scopes associated with authorization tokens, the registry server stores an additional parameter for each authorization token: purpose. From the registry, this is a unique human-readable string that identifies logic run within users' personal data stores. This logic is further described in the question and answer framework section.

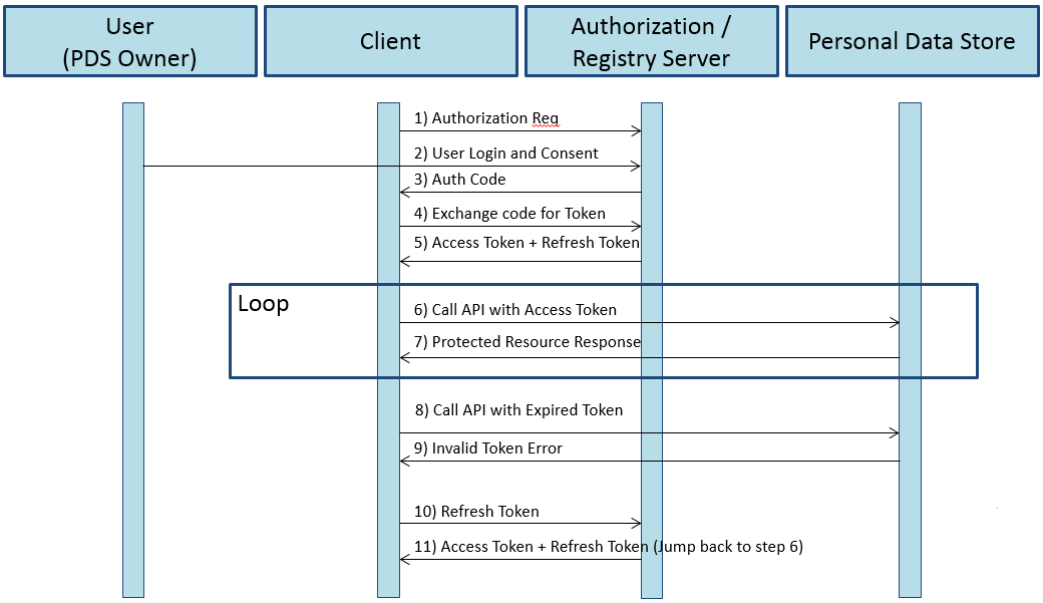

Figure 2: General Authorization Flow - a registered user authorizes an application to act on their behalf. The application then presents an authorization token, which can be refreshed as needed, when accessing the user's data.

For accessing data, the system supports scopes at both the resource and key level. For resource types (such as Funf or Facebook data), a scope named `funf_write` may denote access to write funf data to a user's personal data store. Additionally, to control access to the results of analysis on the raw data within a personal data store, the PDS supports scopes at the key level as well. In this manner, a one-to-one correspondence exists between answers to questions on the PDS and scopes for accessing those questions (a "socialhealth" answer would have a corresponding "socialhealth" scope).

4 Distributed Data Stores

After a user authorizes a third party to store or access their personal data, all subsequent communication is done directly with that user's personal data store using the granted authorization token, as shown in figure 2. Storage for personal data within this location is done in a user-centric manner; while a single openPDS server supports storage and analysis on multiple user's data, each user has a separate backend database, and separate, user-specified encryption keys for all personal data residing in the store, regardless of the data source. In this manner, personal data stores can either have a one-to-one correspondence between physical hosting machines and users, or an arbitrary number of physical machines may provide a logical one-to-one correspondence by sharing physical resources between users. Data from a given user hosted on the same physical machine as other user's may be accessed with a combination of the user's UUID and a valid OAuth token. Figure 3 provides an overview for reference by the subsequent design and implementation sections.

4.1 Storage

In order to support data with an arbitrary schema, MongoDB is the primary backend storage technology. MongoDB is a non-relational ("No-SQL") data store consisting of JSON-formatted documents identified with a unique ID, provided by the MongoDB server running within the PDS. In order to sup-

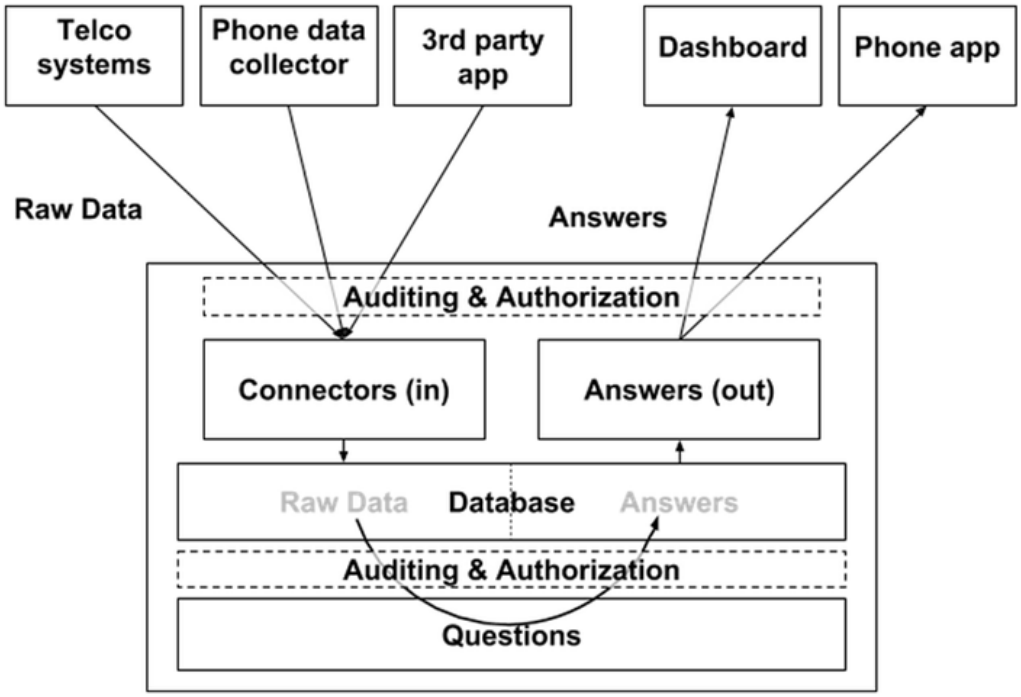

Figure 3: Detailed PDS architecture - Connectors handle data input from external sources, Answers handle output of processed data, Questions handle populating these answers, each step checks for authorization and records an entry in an audit log.

port queries against this data based on time and type of data, each piece of raw data stored in this system has a key denoting the type of data, and a timestamp denoting when the particular sample of data was taken. Given these constraints, relational backends with fixed schemas can also be supported.

To this effect, a SQLInternalDataStore base class has been implemented with corresponding PostgresInternalDataStore and SQLiteInternalDataStore subclasses. Schemas for relational backends are specified in a backend-

agnostic manner in order to support the multitude of different SQL dialects, or other query languages, such as SPARQL. A backend-agnostic schema for a table is represented as a Python dictionary with a name, a list of column tuples containing the name and data type for the column, as well as an optional mapping list for retrieving data from a field that does not match the name provided for the column. An example backend-agnostic schema definition for the CallLogProbe table is provided below:

Listing 1: Example backend-agnostic schema definition.

```
CALL_LOG_TABLE = {
    "name": "CallLogProbe",
    "columns": [
        ("_id", "INTEGER"),
        ("name", "TEXT"),
        ("number", "TEXT"),
        ("number_type", "TEXT"),
        ("date", "BIGINT"),
        ("type", "INTEGER"),
        ("duration", "INTEGER")
    ],
    "mapping": {
        "funf": {
            "number_type": lambda d: d["numbertype"]
        }
    }
```

```
}
```

4.2 Connecting a Data Source

The personal data store currently supports storing data obtained from the Funf framework running on Android phones via a connector. In order to extend the system to support a new type, a connector for that data source must be written, and a user must authorize the new application to access their PDS. These steps are described below.

4.2.1 Authorization to write to the PDS

As all requests to the PDS must contain a bearer token and the UUID of the data store owner as query string parameters (`bearer_token` and `datastore_owner__uuid`, respectively). The endpoint written for the new connector must check for the presence of these querystring parameters and verify them against the registry server that provided them. The endpoint must specify an OAuth scope corresponding to the type of data it is collecting (Funf uses `funf_write`, for example), and the token provided must have access to that scope. Scopes can be created by signed-in users on the registry server at `/admin/oauth2app/accessrange/`.

4.2.2 Extending InternalDataStore

To handle a new data source, the internal API for storing and retrieving data, InternalDataStore, must be updated to handle the new type of data. In most cases, this entails specifying either a new collection in Mongo to hold the data or, for relational backend support, a new SQL table. For relational backends, the schema for the new data type must be provided as described in section 4.1. Authorization against the created scope, connections to the database, and queries against the new collection or table are then provided automatically by the parent InternalDataStore implementation.

4.2.3 Exposing an endpoint for the new connector

After getting authorization to access a PDS, a means of transferring data to it must be provided. This is typically done by exposing a REST endpoint on the PDS for the data source device to POST to. The endpoint will typically accept a POST request of raw data, along with PDS credentials (`datastore_owner__uuid` and `bearer_token`), authenticate the request using the `PDSAuthorization` and `PDSAuthentication` modules, and write the data if the request is valid. For the case of encrypted files, this will also involve a decryption step that can either be done synchronously while the PDS processes the request, or queued up for asynchronous decryption by a separate process.

4.3 A Two-Level API for Accessing Personal Data

As figure 3 shows, accessing data within a personal data store follows a two-step process.

4.3.1 Internal API

As a means of preserving the privacy of the owner, openPDS prohibits data from leaving the store in its raw form. Instead, in order to enforce data use for a specific purpose, the system provides a trusted compute space that a third party may submit code and queries to as a means of computing answers to questions about the data. This compute space is shown within figure 3 as the "Questions" block. Each time raw data is accessed within this space, the data store checks with the registry server to assure the question is allowed access to the raw data it is requesting, and stores an audit log of the request and the code's corresponding purpose. Access to raw data within the trust compute space is provided via the InternalDataStore interface:

class InternalDataStore	
Method	Description
`__init__(profile, token)`	Initializes an InternalDataStore for the user associated with profile, using authorization provided via token.
`getAnswer(key)`	Retrieves a dictionary representing the answer associated with the given answer key, if it exists, or None if it doesn't.
`getAnswerList(key)`	Retrieves a list representing the answer associated with the given answer key, if it exists, or None if it doesn't.
`saveAnswer(key, answer)`	Stores the given answer under the provided answer key.
`getData(key, start, end)`	Retrieves data of the type specified by key, recorded between the start and end times.
`saveData(data)`	Saves data to this InternalDataStore. Data must specify a key and time in order to be accessed later via getData.
`notify(title, content, uri)`	Sends a notification to all devices registered for this user.

Table 1: InternalDataStore provides an interface for accessing raw data and answers within the trusted compute space

4.3.2 External API

Upon accessing the raw data within the question-answering framework, third party code has the ability to store the result of the computation as an answer. These answers provide the second layer of data access on the PDS. While question answering occurs privately within the data store's trusted compute space, answers to questions populated within that space are available via the

answer and answerlist REST APIs on the data store. In order to gain access to the answers stored using an InternalDataStore, a client must list that answer's key as a scope when requesting a bearer token to access the user's PDS. This token, as with all requests against a user's PDS, is included on the request to the answer or answerlist endpoints, where the PDS checks for the requisite scope and stores an audit log of the request prior to returning the pre-computed data. Table 2 documents the full external REST APIs available to clients via standard HTTP requests.

External PDS REST endpoints		
Endpoint	Methods	Description
`/api/personal_data/answer`	GET POST PUT DELETE	Retrieves or stores the answer for a given key as a JSON-formatted dictionary.
`/api/personal_data/answerlist`	GET POST PUT DELETE	Retrieves or stores the answer for a given key as a JSON-formatted array.
`/api/personal_data/funfconfig`	GET POST PUT DELETE	Retrieves or stores Funf configuration objects.
`/api/personal_data/device`	POST DELETE	Registers the cloud-messaging identifier (`gcm_reg_id`) for a client device.
`/api/personal_data/notification`	GET DELETE	Retrieves all outstanding notifications from the PDS.

Table 2: The PDS provides a number of external endpoints for client applications to interface with.

Each REST endpoint in the external API takes `datastore_owner__uuid` and the client's OAuth token as `bearer_token` as querystring parameters. For the case of answer endpoints, the key for the desired answer must also be provided for GET requests. For all endpoints, the authorization layer within the PDS assures that the given bearer token has access to the necessary scopes to complete the request, and an audit log is stored to keep track of the request and the data provided. For the device endpoint, clients with the approved scope can register devices or delete their registration, but cannot perform GET or PUT requests to retrieve or update pre-existing devices. Likewise, the notification endpoint only provides support for retrieving and deleting notifications; notifications may only be added via the internal API.

4.4 Writing Questions

Questions are structured as asynchronous tasks run on a pre-defined schedule within the user's personal data store. The system makes use of Celery, an asynchronous job queue and processing service running continuously within the personal data store's trusted compute space. As such, a question within the personal data store is a Python method that takes an InternalDataStore as parameters and has a `@task()` attribute attached to it. An example task for computing the number of probe entries for a given user is provided below:

Listing 2: Example question for pre-computing an answer from a single user's data.

```
@task()
def recentProbeCounts(internalDataStore):
    start = getStartTime(2, False) #Timestamp from 2 days
        ago
    probes = [
        "ActivityProbe",
        "SimpleLocationProbe",
        "CallLogProbe",
        "SmsProbe",
        "WifiProbe",
        "BluetoothProbe"]
    answer = {}
    for probe in probes:
        #Look up all data for the given probe (no end time)
        data = internalDataStore.getData(probe, start, None
            )
        answer[probe] = data.count()
    internalDataStore.saveAnswer("RecentProbeCounts",
        answer)
```

A file containing each such question is submitted to the personal data store, along with a schedule corresponding to each task that specifies the frequency with which it runs and updates the answer it populates. The personal data store then schedules each task to run and allows each to only update answers with keys that the provided token has specified as scopes. For the case of a single personal data store hosting a number of user's data, the data store iterates over all users hosted on the given PDS install and runs

each question installed for that user before moving on to the next user. In this manner, the backend storage can take advantage of time locality between references to similar data in order to maximize the amount of data present in the backend storage's cache, if it exists.

4.5 Group Computation

Data and answers about that data, are often more useful when aggregated across different users. However, the ability to perform such aggregation is notably absent from the aforementioned question answering code. In order to perform an aggregation across different users' personal data stores, a second question answering format is specified, along with a number of endpoints within the PDS to facilitate group computation across a distributed set of machines.

As with the single-user case, questions that contribute to group answers are asynchronous celery tasks. However, instead of being run within the PDS continuously on a set schedule, these tasks are invoked via a group contribution URL on the user's PDS, located at `/aggregate/contribute`. Personal data stores POST to this endpoint in a ring fashion to build up a running total for the aggregate computation. This endpoint takes the UUID of the user contributing to the computation as a querystring parameter, and the body of the POST request contains both the current intermediate result of the running computation as well as the remaining list of personal data stores to visit in order to complete the computation. Each personal data

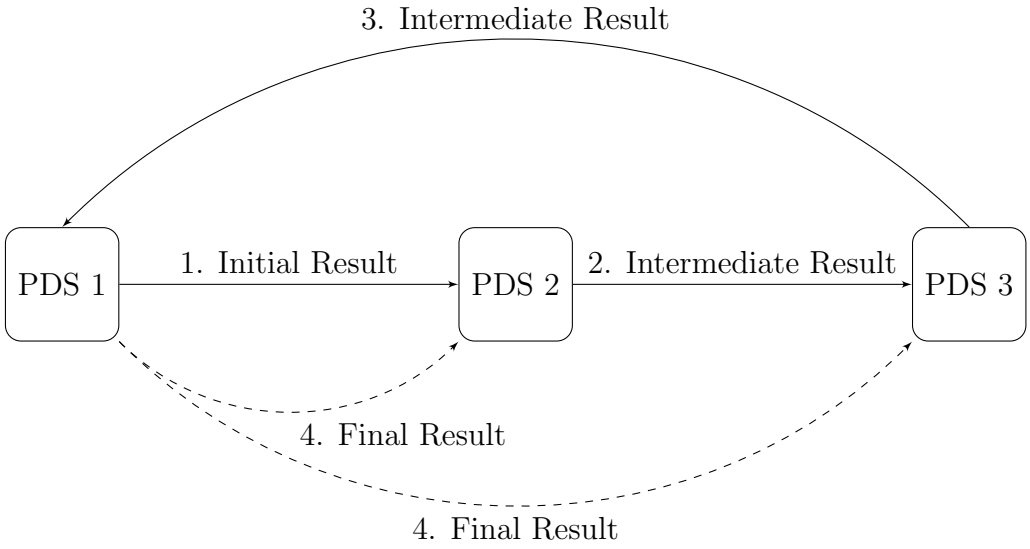

Figure 4: Group computation flow: 1) Initiating PDS begins aggregate computation 2-3) PDSes contribute to intermediate results passed along the chain of contributers 4) The initiating PDS completes the computation and POSTs the result back to all contributing data stores

store updates the running total, taking into account data within that PDS, and then posts the updated intermediate result to the next personal data store in the chain. Figure 4 describes the flow of data amongst a small set of contributing personal data stores.

To take the running total into account when contributing to a group computation, the format for such questions modifies the original question format to take the current running total in addition to an InternalDataStore. Below is an example of such a question contribution that leverages the probe counts answer calculated by the previous example question:

Listing 3: Example question for contributing to an aggregate computation.

```
@task()
def contributeToAverageProbeCounts(ids, runningTotal):
    myCounts = ids.getAnswer("RecentProbeCounts")
    count = runningTotal["count"]
    for probe in myCounts:
        runningTotal[probe] = runningTotal[probe]*count
        runningTotal[probe] += myCounts[probe]
        runningTotal[probe] = runningTotal[probe]/(count+1)
    runningTotal["count"] += 1
    return runningTotal
```

After calling the method associated with a given aggregate question, the PDS takes the return value and POSTs it, along with the list of remaining contributers to the next PDS in the chain. When the running computation reaches the initiating PDS, a separate completion method is called that pushes the final result out to each contributing PDS, where a copy of the result is stored locally.

4.6 Aggregate Performance and Approximations

Aggregate computations incur a latency cost over traditional methods that would pull raw data for all contributing parties to a centralized place and perform computation therein. For a small sample population of hundreds of users, this latency can be acceptable - the set of users a registry server returns for a given computation can be ordered in a way that clusters personal data stores in similar geographic locations as a best-effort means of minimizing

latency of the total computation. Assuming average latency between data stores can be kept below 500ms, every 100 separate data stores will add at most 50 seconds of latency to the computation. This is deemed acceptable for the typical group computation workload, where the value of the aggregate changes smoothly and slowly over time and thus, computing the aggregate in an offline manner and storing the result is acceptable with minimal loss of accuracy over time.

However, in the presence of even thousands of separate personal data stores, latency can begin to affect the accuracy of the computation. Table 3 shows computation times and overhead for a typical group aggregate that computes the average and standard deviation of a given answer as the number of personal data stores grows. These numbers were taken from a simulated workload between 2 personal data stores with identical system configurations (1.6 GHZ dual-core processor, 2GB RAM), and 40ms of network latency between them, as measured by `ping`.

Data stores	Computation Time (s)	Overhead (s)	Total Time (s)
1	0.011	0	0.011
2	0.022	0.27	0.292
5	0.05	1.43	1.48
10	0.16	2.66	2.82
100	1.64	28.4	30.2
1000	15.1	302.27	317.37
10000	156.2	3166.215	3316.415

Table 3: Aggregate performance and overhead - Overhead quickly overtakes computation time when computing an aggregate.

For the purposes of analysis, a result that takes less than 5 seconds to compute will be considered feasible for real-time computation at the time of the request, and any result that takes less than 5 minutes to compute will be considered "real-time" if the pre-computed result is stored in a manner that can be quickly retrieved. This method of retrieving an up-to-date, but pre-computed result at the time of the request for an answer is known as a "real-time batch", where the result is returned in real-time (the time it takes to look up an answer in a PDS is less than 100ms), and the data has been calculated based on the most up-to-date raw data available. As the number of data stores contributing to a computation grows, time to complete one rotation around all contributing data stores makes real-time results intractable for anything more than 10 contributing data stores. Furthermore, when 1000 or more data stores contribute to a result, even maintaining a real-time batch result becomes unfeasible.

To circumvent this limitation, personal data stores have the ability to short-circuit aggregate computation by returning an intermediate result to the data store that initiated the computation, allowing it to push out the partial result of the computation to contributing data stores prior to completing the computation. The number of data stores to visit prior to storing an intermediate result is configurable on a per-question basis. For the provided workload, a good time to store an intermediate result might be after visiting 100 data stores. Visiting the contributing personal data stores in an unbiased order allows the system to store the intermediate result of the

5-minute long computation in 30 seconds with 10% confidence. For each 100 contributions after the approximate result is computed, the resulting approximate answer is updated with an additional 10% confidence, until the computation is complete and the result represents the true value.

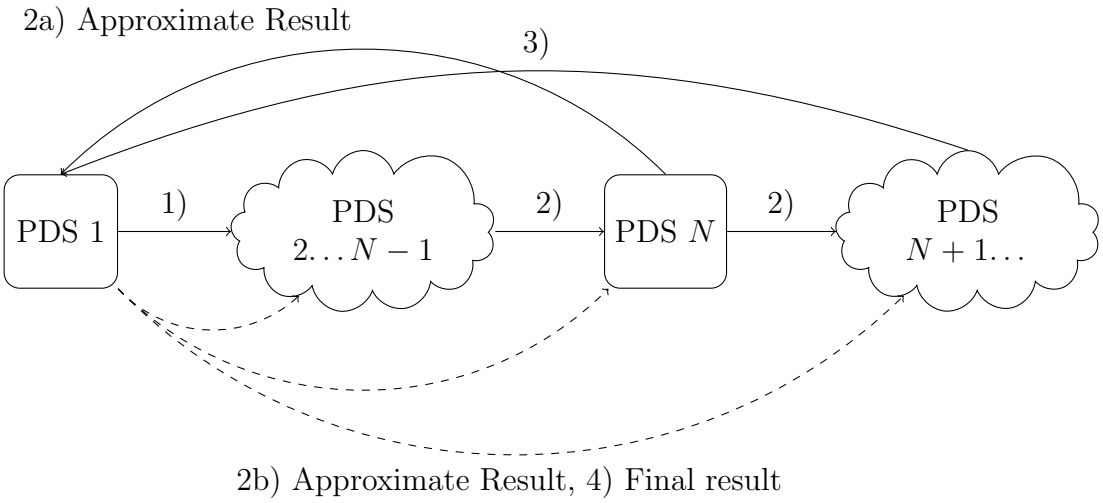

Figure 5: Group approximation flow: 1-2) PDS's contribute to intermediate results passed along the chain of contributers 2a) After N contributions have been made, an approximate result is returned to the initiating PDS 2b) The initiating PDS pushes this result to all contributors 2-3) Aggregation continues, returning approximate answers ever N contributions, until 4) the final result is returned.

The ability to return approximate results means that each contribution to a running calculation must be representative of the final result - if the final result of an aggregate computation depends on post-processing within the initiating data store, prior to pushing to the contributing stores, this post-processing must apply to an intermediate (approximate) result as well. To facilitate such post-processing, the personal data store keeps track of the

number of contributers along with the running total passed between contributers. Under the approximate flow, contributers to the example calculation would receive an intermediate result approximately every 30 seconds with increasing confidence until the 5 minute computation has completed.

4.7 Clustered Hosting

Using the group approximation framework, there is the potential to further enhance performance by clustering a number of logical personal data stores within a single physical machine, and contributing to group computations on all clustered data stores at once. It is important to note that this is different than hosting all personal data stores in an ecosystem in the same place - in an ecosystem of thousands or millions of personal data stores, such an approach would provide a single point of attack and furthermore, is prone to leaking everyone's data in the event of a security breach. Rather, from the aggregate performance numbers above, a good initial guess at the number of personal data stores to host within a single machine would be 100, resulting in reasonable performance, and little to no increase in the likelihood of discerning which machine to attack for a given user's data.

For the case of a clustered personal data store that represents the data from a number of different users, the personal data store framework can reorder the list of contributers for a given aggregate computation to pull out all users that are hosted within the given physical machine, and perform a portion of the aggregate computation entirely within the machine before

passing the intermediate result along to the next contributer. In this manner, figure 5 stands as an accurate representation of the flow, however the clouds of PDSes in the figure could represent either N physically separate personal data stores, or N clustered personal data stores within a single physical machine. Table 4 shows the results of the same computation as table 3, with clusters consisting of 100 logical personal data stores. While overhead is decreased by a factor of 100, computation sees a commensurate decrease, likely due to better resource utilization within the cluster (code can stay in memory, personal data can stay in cache).

Data stores	Computation Time (s)	Overhead (s)	Total Time (s)
1	0.01	0	0.01
10	0.1	0	0.16
100	0.55	0	0.55
200	3.22	0.27	3.49
1000	4.17	2.66	6.83
10000	40.37	28.4	68.77

Table 4: Clustered aggregate performance - Overhead is reduced by a factor of N for clusters of N personal data stores. Furthermore, computation time decreases as cache and memory within a cluster is better utilized.

As aggregation within a clustered personal data store occurs without even intermediate aggregate results leaving the store, clustered hosting also has the potential to further preserve the privacy of individuals participating in group aggregate computations; an individual's personal data store within a cluster becomes more resilient to attacks from outside the cluster that would attempt to utilize the group computation framework to re-identify the

individual based on their contributions to group computations.

5 Client Applications

To facilitate data collection and user interaction, a Java client library is implemented on top of the Android platform. In addition to extending Funf data collection to incorporate access control and syncing both data and configuration with a personal data store, classes are provided to interface directly with both the registry and distributed data store components of the openPDS personal data ecosystem.

Applications must follow the authentication flow described in figure 2, which, from the client library perspective, involves the use of two library classes: RegistryClient and PersonalDataStore. These classes wrap corresponding interactions with the Registry Server for authentication and authorization, and the PersonalDataStore for retrieving data and visualizations. The RegistryClient and PersonalDataStore interfaces are provided below in tables 5 and 6.

A typical client application will first prompt the user to either register or login, and will then retrieve authorization and a link to the user's personal data store. To this end, the openPDS client library provides a number of convenience asynchronous tasks: `UserLoginTask`, `UserRegistrationTask`, and `UserInfoTask` that wrap each of the corresponding methods on RegistryClient, and update the application's SharedPreferences to store the nec-

class RegistryClient		
Method	Arguments	Description
RegistryClient	String url String clientKey String clientSecret String basicAuth	Constructs a RegistryClient connecting to the given url for the given client credentials.
authorize	String username String password	Attempts to authorize the client associated with this RegistryClient to access the given user's account.
getUserInfo	String token	Attempts to retrieve information, including PDS location and UUID for the user that issued the provided token.
createProfile	String email String password String firstName String lastName	Registers a new user with the given credentials on the Registry server.

Table 5: RegistryClient provides a means of interfacing with a Registry server from an Android client application.

essary information used to construct a PersonalDataStore.

After the user has registered or logged in via the RegistryClient, and has authorized the client application to access their personal data store, all subsequent communication is done directly via PersonalDataStore objects (described in table 6). A client application can request answers from the PersonalDataStore object and present native user interface widgets, or request a fully-qualified URL from a PersonalDataStore object and display a Webview containing the corresponding page on the user's personal data store.

class PersonalDataStore		
Method	Arguments	Description
PersonalDataStore	Context context	Initializes a PersonalDataStore given an Android Context with the requisite information (PDS location, user UUID, bearer token).
getAnswer	String key String password	Retrieves a JSONObject for the answer corresponding to the given key from the PDS if it exists and the client has access to it. Null otherwise.
getAnswerList	String key String password	Retrieves a JSONArray for the answer corresponding to the given key from the PDS if it exists and the client has access to it. Null otherwise.
registerGCMDevice	String regId	Registers the current device with the provided GCM registration ID on the PDS.
getNotifications		Retrieves all notifications on the PDS
buildAbsoluteUrl	String relativeUrl	Builds the absolute URL for the resource located at relativeUrl on this PDS.

Table 6: PersonalDataStore provides a means of interfacing with a user's Personal Data Store from an Android client application.

5.1 Configuring Data Collection

The openPDS client extends Funf data collection by providing an implementation of the standard Funf Pipeline interface named OpenPDSPipeline. This is used in place of Funf's BasicPipeline in the configuration file for Funf and, as with standard Funf configurations, the configuration is a JSON-formatted

dictionary containing a `@type` field for specifying the fully qualified pipeline class, followed by a dictionary for declaring scheduled actions, and an array for declaring the data the pipeline instance will collect. An example configuration file for configuring data collection from call logs and location are provided below:

Listing 4: Example openPDS Funf pipeline definition.

```
{
  "@type": "edu.mit.media.openpds.client.funf.
     OpenPDSPipeline",
  "schedules": {
    "upload": {"strict": false, "interval": 900 }
  },
  "data": [
    {
      "@type": "edu.mit.media.funf.probe.builtin.
         CallLogProbe",
      "afterDate": 1365822705,
      "@schedule": {"strict": false,"interval": 3600 }
    },
    {
      "@type": "edu.mit.media.funf.probe.builtin.
         LocationProbe",
      "maxWaitTime": 30, "goodEnoughAccuracy": 10,
      "@schedule": {"strict": true,"interval": 900 }
    }
```

```
  ]
}
```

This configuration is provided as metadata within the service tag for the FunfManager service in the application's AndroidManifest.xml file. Additionally, permissions for each probe must be requested within this file as well, including fine and coarse grain location for any location probe, the requisite admin permissions for wifi and bluetooth scanning probes, and external storage and internet permissions for accessing the user's personal data store and writing data.

5.2 Defining Client Data Use and Visualizations

The full definition for a client application built on top of the openPDS ecosystem consists of three parts:

1. A client manifest file
2. A file containing all questions the app needs
3. A set of HTML visualization files

Each of the above requirements are combined into a zip archive to be installed on a user's personal datas tore. This section describes how each of these are defined and used by the openPDS client library and servers to create a cohesive client application.

5.2.1 Client Manifest

An client manifest file format is specified as a means of defining how a client application may present answers to questions about data, as well as what data contributes to a particular answer. Specifying a manifest for an app built on top of the openPDS ecosystem allows the client library to automatically generate user interfaces for notifying and providing control to the user over how their data is collected at the client level. The app manifest file is a JSON-formatted dictionary containing fields for the app name, client credentials, visualizations, and any answers the app will request access to, along with the data required to compute each answer. An example is provided below:

Listing 5: Example client manifest definition.

```
{
  'name': 'App Name',
  'credentials': {
    'key': 'registered_client_key',
    'secret': 'registered_client_secret',
    'auth': 'BASIC auth_string_here',
    'scopes': [ 'funf_write', 'answer1', 'answer2', '
       answer3']
  },
  'visualizations' : [
    { 'title': 'Visualization 1 Title', 'key': 'vis1Key',
      'answers': ['answer1'] },
```

```
      { 'title': 'Visualization 2 Title', 'key': 'vis2Key',
        'answers': ['answer2', 'answer1'] }
    ],
    'answers': [
      { 'key': 'answer1',
        'data': ['answer3', {'key':'WifiProbe','required':
           false]}},
      { 'key': 'answer2',
        'data': ['ActivityProbe', 'LocationProbe'] },
      { 'key': 'answer3',
        'data': ['SmsProbe','CallLogProbe','BluetoothProbe']
           }
    ]
}
```

The client library constructs access control screens for each answer based on the data it requires. For compound answers - those that rely on the result of another answer - the client library traverses the resulting data dependency tree to map back to the necessary probes required for computing the answer. Additionally, the client library provides a suite of user interface elements for displaying server-generated visualizations from the visualization keys provided in the client manifest. A well-defined client manifest file, combined with the pipeline definition metadata, allows a developer building on top of the personal data ecosystem to quickly develop applications with minimal client-side code, if they so choose, and have the client library automate the process of authorization, access control, and UI generation for visualizations.

5.2.2 Questions File

Each client must define any new questions they would like to assure will be populated on the user's personal data store. Individual questions are to be written as described in section 4.4, and included in a single `questions.py` file. If the client application requires questions from another application, the unique identifier for that question's key must be included in the client manifest file, but the definition of the question need not be included in the questions file - the given client will simply have a hard dependency on the related client being installed on the user's PDS.

5.2.3 Visualizations

As defined in the client manifest file, an openPDS client application can specify a number of visualization keys to automatically generate visualizations that have been approved and installed on a user's personal data store. These visualizations are available on the personal data store at `/visualization/key`, and take `bearer_token` and `datastore_owner` as querystring parameters.

As visualizations are generated on the user's personal data store, standard web languages are used to write them - HTML, Javascript, and CSS. To facilitate writing visualizations in web languages, the personal data store provides a set of Javascript libraries for interfacing with the endpoints described in 4.3.2. These includes answer and answerlist libraries, which provide Backbone.js collections and models for the corresponding objects in the external PDS API. Developers then write corresponding Backbone.js Views to pull in

data from the provided collections and models and present the user with a visualization layer. Standard supporting libraries, such as jQuery and jQuery Mobile are provided by default on all visualization files, and externally-hosted Javascript and CSS files may also be imported.

Visualizations are built on top of Django's template language for HTML and consist of three template blocks a developer must fill in: `title`, `scripts`, and `content`. The personal data store uses these three sections to fill in a complete HTML page for the visualization. An example visualization file is provided below:

Listing 6: Example visualization file

```
{% block title %}
Recent Places
{% endblock %}
{% block scripts %}
<script src="//openlayers.org/dev/OpenLayers.mobile.js"></
    script>
<script src="{{ STATIC_URL }}js/answerList.js"></script>
<script src="{{ STATIC_URL }}js/answerListMap.js"></script>
<script>
$(function () {
    window.answerListMap = new AnswerListMap("RecentPlaces"
        ,"answerListMapContainer");
    $(window).bind("orientationchange resize pageshow",
        answerListMap.updateSize);
```

```
        $("#plus").live('click', answerListMap.zoomIn);
        $("#minus").live('click', answerListMap.zoomOut);
    });
    </script>
    {% endblock %}
    {% block content %}
    <div id="answerListMapContainer">
    </div>
    <div id="footer" data-role="footer">
    </div>
    {% endblock %}
```

In the above example, HTML is used only for structure on the page. Resulting information is filled in as a map using `answerListMap.js` - a pre-defined View provided on openPDS for displaying answers with latitude and longitude components on a map. When the key for this visualization is included in a client manifest, the client application will load a Webview with the corresponding fully-qualified URL to reach this visualization in the app, allowing an authorized app to display the visualization without supporting Java code.