Distributed Aggregation Protocol (DAP) Primer

Abhishek Tiwari

Citation: *A. Tiwari*, ”Distributed Aggregation Protocol (DAP) Primer”, Abhishek Tiwari, 2024. doi:10.59350/bd30z-hxs16

Published on: October 02, 2024

In last post we covered, Privacy Preserving Measurement (PPM) and discussed how Distributed Aggregation Protocol (DAP) works (see [1]). Today, we'll explore how to implement a simplified version of the DAP using Python with Prio3 as our Verifiable Distributed Aggregation Function (VDAF). This implementation will support multiple clients, demonstrating how DAP can aggregate data from multiple sources while maintaining privacy. Let's dive in!

Implementing the Finite Field

First we will sets up a simple Finite Field (FF), which is crucial for the cryptographic operations in Prio3. A Finite Field is a set that provides a way to perform arithmetic operations with guaranteed properties that are useful for cryptographic protocols. In Prio3, Finite Fields are used for secret sharing and for the underlying zero-knowledge proofs. Prio3 is suitable for a wide variety of aggregation functions, including sum, mean, standard deviation, estimation of quantiles, and linear regression.

Here we setup addition, multiplication, subtraction, and division operations on top of Finite Field. Initialisation requires a prime number p. We're using largest 32-bit prime (4294967291), larger the prime better the security.

```
import random

class FiniteField:
    def __init__(self, p):
        if not self._is_prime(p):
            raise ValueError("p must be prime")
        self.p = p

    def add(self, a, b):
        return (a + b) % self.p

    def sub(self, a, b):
        return (a - b) % self.p

    def mul(self, a, b):
        return (a * b) % self.p

    def div(self, a, b):
        if b == 0:
            raise ZeroDivisionError("Cannot divide by zero in the field")
        # Use Fermat's little theorem to compute the multiplicative
           inverse
        return self.mul(a, pow(b, self.p - 2, self.p))

    def _is_prime(self, n):
        if n < 2:
            return False
        for i in range(2, int(n ** 0.5) + 1):
```

```
            if n % i == 0:
                return False
        return True

# Use a larger prime for better security
# This is the largest 32-bit prime
FIELD = FiniteField(4294967291)

# Example usage:
a = random.randint(0, FIELD.p - 1)
b = random.randint(0, FIELD.p - 1)

print(f"a = {a}, b = {b}")
print(f"a + b = {FIELD.add(a, b)}")
print(f"a - b = {FIELD.sub(a, b)}")
print(f"a * b = {FIELD.mul(a, b)}")
print(f"a / b = {FIELD.div(a, b)}")
```

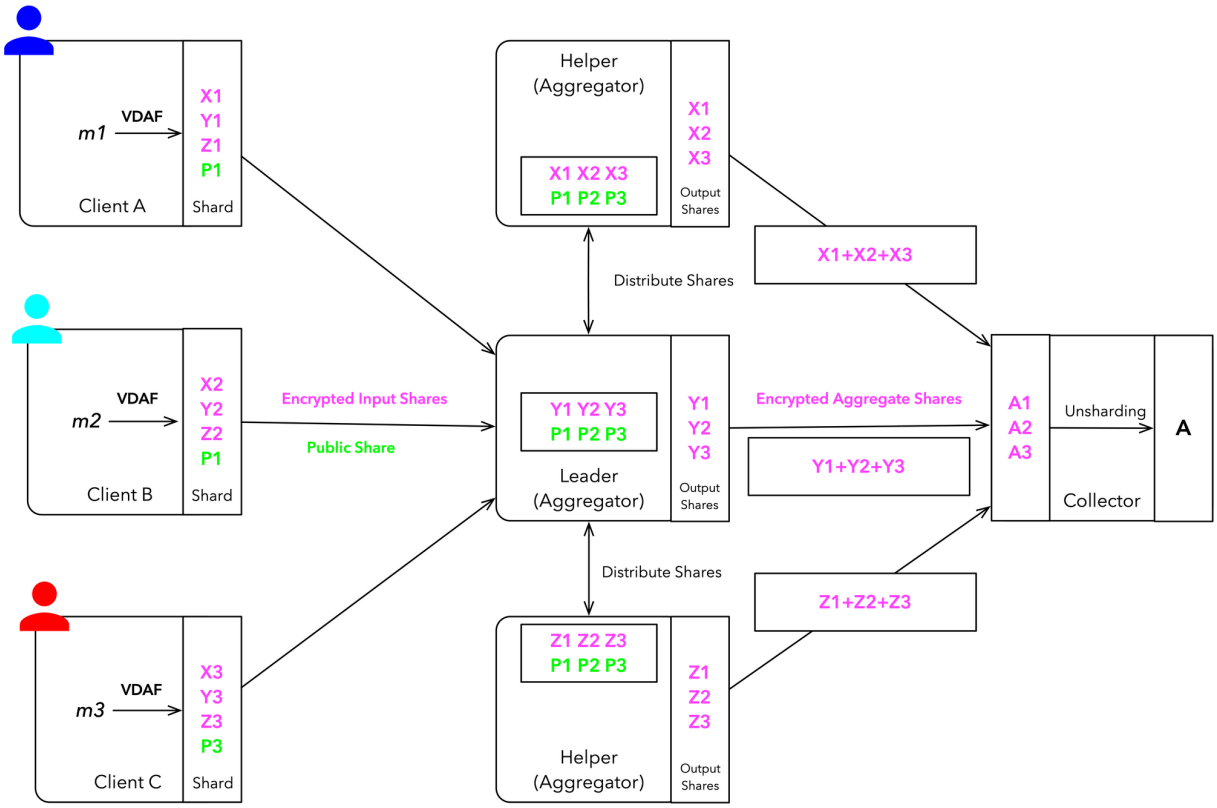

Figure 1: DAP Roles - Clients generate the original measurement data, Aggregators participate in multi-party aggregation, and Collector receives the final aggregated, privacy-preserving results.

Implementing Prio3

This is a simplified implementation of Prio3 which requires a FF for initialisation. We define 4 methods: shard, prepare, aggregate, and unshard.

The shard method splits a measurement into two shares, prepare is simplified to an identity function, aggregate sums the shares, and unshard combines the final aggregate shares.

```
class Prio3:
    def __init__(self, field):
        self.field = field

    def shard(self, measurement, nonce):
        # Ensure the measurement is in the field
        measurement = measurement % self.field.p
        # Generate a random share
        share1 = random.randint(0, self.field.p - 1)
        # Compute the second share such that share1 + share2 = measurement
            (mod p)
        share2 = self.field.sub(measurement, share1)
        return None, [share1, share2]

    def prepare(self, agg_id, agg_param, nonce, public_share, input_share)
       :
        # In this simple version, preparation is just the identity
           function
        return input_share

    def aggregate(self, agg_param, output_shares):
        # Sum up the output shares in the field
        return sum(output_shares) % self.field.p

    def unshard(self, agg_param, agg_shares, num_measurements):
        # Sum up the aggregate shares in the field
        return sum(agg_shares) % self.field.p

vdaf = Prio3(FIELD)

# Example usage:
measurement = 12345
nonce = os.urandom(16)
public_share, input_shares = vdaf.shard(measurement, nonce)

print(f"Original measurement: {measurement}")
print(f"Shares: {input_shares[0]}, {input_shares[1]}")
print(f"Sum of shares: {(input_shares[0] + input_shares[1]) % FIELD.p}")
```

Defining DAP Structures

These classes define the main data structures used in DAP: reports, aggregation jobs, and collections.

```
class Report:
    def __init__(self, report_id, time, public_share,
       encrypted_input_shares):
        self.report_id = report_id
        self.time = time
        self.public_share = public_share
        self.encrypted_input_shares = encrypted_input_shares

class AggregationJob:
    def __init__(self, job_id, agg_param, reports):
        self.job_id = job_id
        self.agg_param = agg_param
        self.reports = reports

class Collection:
    def __init__(self, report_count, interval, encrypted_agg_shares):
        self.report_count = report_count
        self.interval = interval
        self.encrypted_agg_shares = encrypted_agg_shares
```

Implementing the Client

The Client generates reports by sharding measurements and simulates uploading them to the Leader.

```
class Client:
    def __init__(self, leader_url, helper_url, task_id):
        self.leader_url = leader_url
        self.helper_url = helper_url
        self.task_id = task_id

    def generate_report(self, measurement):
        report_id = os.urandom(16)
        timestamp = int(time.time())
        public_share, input_shares = vdaf.shard(measurement, report_id)
        encrypted_shares = input_shares
        return Report(report_id, timestamp, public_share, encrypted_shares
            )

    def upload_report(self, report):
        print(f"Uploading report {report.report_id.hex()} to Leader")
        return True
```

Implementing the Leader

The Leader receives reports from clients, starts aggregation jobs, and processes its share of the reports.

```
class Leader:
    def __init__(self, helper_url, collector_public_key):
        self.helper_url = helper_url
        self.collector_public_key = collector_public_key
        self.reports = {}
        self.aggregation_jobs = {}

    def receive_report(self, report):
        self.reports[report.report_id] = report
        return True

    def start_aggregation_job(self, job_id, agg_param, report_ids):
        job = AggregationJob(job_id, agg_param, [self.reports[rid] for rid
            in report_ids])
        self.aggregation_jobs[job_id] = job

        output_shares = []
        for report in job.reports:
            input_share = report.encrypted_input_shares[0]  # Leader's
                share
            output_share = vdaf.prepare(0, agg_param, report.report_id,
                report.public_share, input_share)
            output_shares.append(output_share)

        agg_share = vdaf.aggregate(agg_param, output_shares)
        print(f"The aggregate share of leader is: {agg_share}")

        return agg_share
```

Implementing the Helper

The Helper processes its share of the reports in each aggregation job.

```
class Helper:
    def __init__(self, collector_public_key):
        self.collector_public_key = collector_public_key

    def process_aggregation_job(self, job):
        output_shares = []
        for report in job.reports:
            input_share = report.encrypted_input_shares[1]  # Helper's
                share
```

```
            output_share = vdaf.prepare(1, job.agg_param, report.report_id
                , report.public_share, input_share)
            output_shares.append(output_share)

        agg_share = vdaf.aggregate(job.agg_param, output_shares)
        print(f"The aggregate share of helper is: {agg_share}")

        return agg_share
```

Implementing the Collector

The Collector requests and processes the final aggregate result.

```
class Collector:
    def __init__(self, leader_url, private_key):
        self.leader_url = leader_url
        self.private_key = private_key

    def request_collection(self, task_id, batch_interval, leader_agg_share
        , helper_agg_share):
        print(f"Requesting collection for task {task_id}")

        encrypted_agg_shares = [leader_agg_share, helper_agg_share]

        collection = Collection(len(leader.reports), batch_interval,
            encrypted_agg_shares)
        return self._process_collection(collection)

    def _process_collection(self, collection):
        decrypted_shares = collection.encrypted_agg_shares

        result = vdaf.unshard(None, decrypted_shares, collection.
            report_count)
        return result
```

Running the DAP End-to-end

Finally, we will sets up the DAP participants, creates multiple clients, generates and processes reports, and computes the final aggregate result.

```
# Set up the DAP participants
leader = Leader("http://helper.example", "collector_public_key")
helper = Helper("collector_public_key")
collector = Collector("http://leader.example", "collector_private_key")

# Create multiple clients and generate reports
num_clients = int(input("Enter the number of clients: "))
```

```
clients = [Client("http://leader.example", "http://helper.example", "
    task123") for _ in range(num_clients)]

for i, client in enumerate(clients):
    measurement = int(input(f"Enter measurement for client {i+1}: "))
    report = client.generate_report(measurement)
    client.upload_report(report)
    leader.receive_report(report)

# Leader starts an aggregation job
job_id = os.urandom(16)
agg_param = None  # Not used in our simple Prio3 implementation
leader_agg_share = leader.start_aggregation_job(job_id, agg_param, [report
    .report_id for report in leader.reports.values()])

# Helper processes the aggregation job
helper_agg_share = helper.process_aggregation_job(leader.aggregation_jobs[
    job_id])

# Collector requests and processes the collection
batch_interval = (int(time.time()), 3600)  # Last hour
result = collector.request_collection("task123", batch_interval,
    leader_agg_share, helper_agg_share)

print(f"The aggregate result (sum of all measurements) is: {result}")
print(f"The average of the measurements is: {result / num_clients}")
```

Here is example output from the implementation,

```
Enter the number of clients:  3
Enter measurement for client 1:  19
Uploading report 8dcc5a57ed3f06a29c25ef02194d01ce to Leader
Enter measurement for client 2:  11
Uploading report 09249da048604887b50d0d95022847d4 to Leader
Enter measurement for client 3:  15
Uploading report 0f7f4bc8fd3f9724e7559d9184c4c869 to Leader
The aggregate share of leader is: 2424286919
The aggregate share of helper is: 1870680417
Requesting collection for task task123
The aggregate result (sum of all measurements) is: 45
The average of the measurements is: 15.0
```

Conclusion

While this implementation is greatly simplified, it demonstrates the core concepts of DAP. In a production DAP system, each step would involve complex cryptography to ensure security and privacy. The system would also include features like batch processing of reports, minimum batch sizes to enhance privacy, various query types for different aggregation needs, robust error handling and security mea-

sures, etc. For production grade, implementation please refer libprio-rs from Divvi Up, daphne from Cloudflare, prio from original creators of Prio system and finally janus - an experimental implementation of the DAP - from Divvi Up.

References

[1] A. Tiwari, “Privacy Preserving Measurement,” 2024, *Abhishek Tiwari*. doi: 10.59350/fnpfz-3v466.