A System Design for Privacy-Preserving Reach and Frequency Estimation

Craig Wright, Raimundo Mirisola, Jason Frye, Sanjay Vasandani, Mark Fashing, Eli Fox-Epstein, Yunus Yenigor, Yao Wang

Objective

This document describes the high level system design of an MPC-based approach to privacy-preserving reach frequency estimation. The MPC protocol was previously described in Privacy Preserving Secure Cardinality and Frequency Estimation [1], and although several modifications are forthcoming, they do not impact the overall system design.

Background

In [1] we outlined two methods for computing reach, which can be broadly classified into two families. The first family of methods require that publishers apply differentially private noise directly to their sketches when they are created. This ensures that all publisher outputs are private and can be combined in any way a client sees fit. However, as the number of sketches being combined increases, the noise compounds and utility is diminished. We have also found this method to be completely unworkable for frequency estimation.

On the other hand, the second family of methods requires that publishers provide encrypted sketches to a secure multiparty computation (MPC) platform, which then combines the sketches without ever decrypting them and only computes a single noisy reach and frequency estimate. The main features of the MPC-based approach are that:

1. It computes its output solely on encrypted inputs, which remain encrypted even while being computed upon, and only agreed upon outputs are ever revealed. Moreover, such outputs can be made differentially private as part of the encrypted work load.
2. Its error for both reach and frequency computation is constant regardless of the number of inputs, which means it can scale to any number of data providers without losing any accuracy.
3. Its security model is distributed, which means that:
    a. There is no single trusted third party that must be relied upon for either trustworthiness of output or security of data.
    b. Only explicitly requested outputs are computed and there will be several independent logs of these.
    c. The only information learned from running the system is the agreed upon outputs of the system
    d. Compromising a single component of the system will not allow an attacker to learn anything about data provider inputs. Indeed all components of the system would need to be compromised in order for any of the encrypted inputs to be revealed, which is a huge advantage in a world where high profile data breaches seem all too common.

A downside to this approach is that it will be more expensive, both to build and to operate, raising an important question about whether a MPC-based R/F estimator can scale to meet the demands of our cross media measurement use cases while still being cost effective. We do however, believe that MPC-based methods have significant upside, both in terms of privacy and accuracy, for this and other x-media measurement use cases like multi-touch attribution (MTA) and sales lift. Therefore we have recently begun building this system in order to determine whether it is able to effectively meet the demands of our use cases.

This document continues by reviewing the requirements for the cross media reach and frequency Measurement and then goes on to describe the system architecture of an MPC-based reach and frequency estimator that is powered by the algorithms described in [1]. In particular it considers implementing the MPC protocol for the Liquid Legions cardinality estimator (section 5 of [1]). Familiarity with the protocol described in the above paper is assumed, but is not strictly required to understand this document.

Overview of Requirements

The system described below focuses on computing reach and frequency estimates for x-media advertisements and content. However, in addition to this basic functional requirement there are a set of non-functional requirements that should also be kept in mind.

1. In order to take advantage of the security model provided by MPC, there must be:[1]
    a. at least two independently operated MPC nodes and exactly one coordinator per deployment;
    b. at least one MPC node must run on a different cloud than the others; and
    c. the coordinator should be operated independently of the MPC nodes.
2. The system must scale to accommodate ads traffic for any media market, and should be able to scale to support global ads measurement traffic.
3. It must be reliable and highly available.
4. It must be easy to integrate with for both Clients (advertisers, agencies, etc.) and Data Providers alike.
5. It must effectively utilize computational resources. In particular, due to the large number of cryptographic operations and large data payloads, efficiency of cryptographic operations and avoidance of unnecessary network communication are especially important.
6. It should be implemented with best-in-class, cloud agnostic, open-source tooling and infrastructure technology.
7. It must be architected to be runnable on multiple cloud environments, and minimize the reliance on proprietary APIs. Where this is not possible suitable wrappers should be utilized.
8. Technology and design choices should enhance long-term developer productivity and system maintainability.
9. System interactions and dependencies should be made as simple as possible. This includes a preference for microservices, database isolation, and a separation of internal APIs from external APIs.

Design Details

A cross-media reach and frequency measurement system (XMMS) should enable advertisers and eventually content owners to compute reach and frequency measurements across creative content, publishers, and media types while preserving user privacy. At the highest level it is composed of three components - Data Provider APIs, Client APIs, and the implementation of the estimator itself. We have chosen to use the term “Client” to encompass both advertisers and

[1] Overall this document does not dwell on security, but is focused on overall data flow and those aspects that are required to evaluate the cost effectiveness of the system.

content owners, because even though advertisers are our initial focus, there is nothing in this design that precludes it from being used to measure content in the future.

The following sections introduce these three components and discuss the major subcomponents of each as well as the relationships between the various subcomponents. However, this is still a rather high level view of the overall system architecture. In particular this document does not consider the fine details of any API, any schema whatsoever, or the details of the deployment and test infrastructure required to actualize this system. Notably it also omits details about how to secure this system. These omissions are not meant to imply that any of these topics are unworthy of concern, rather that introducing the overall flow of data in the system is the primary concern at this time. The omitted items will be discussed in subsequent documents.

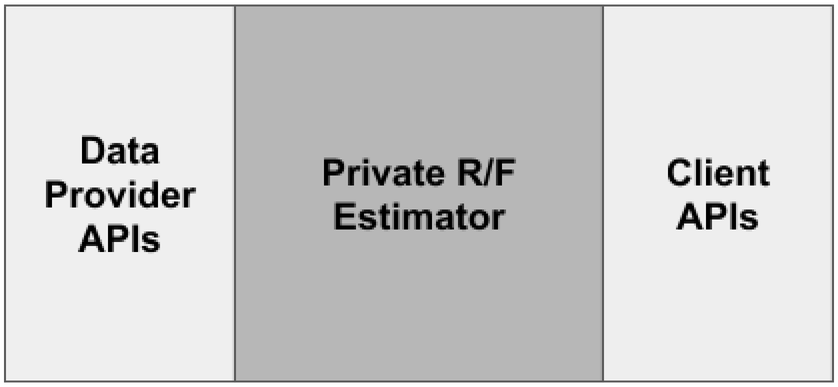

Fig: Three Major Components of Private Reach and Frequency Estimator

Tracking Content and Ad Campaigns

In order to take into account both ads and content measurement we have chosen a broader term, Measurable Unit to refer both to ad campaigns and content measurement. The only thing that must be agreed upon is a universal measurement client ID (UMCID), which the measurement system itself will issue upon client registration, where a client is defined as an advertiser or content measurement consumer. A client would then provide this ID to impression data providers who would then use it to tag all of the campaigns/content associated with that client.

Subsequently each data provider will register a client's measurable units with the XMMS by providing their data provider ID, the appropriate UMCID, an arbitrary measurable unit ID, and additional metadata. Once this occurs, clients can search for their measurable units using the XMMS Search Service. Finally, once a client has found their measurable units they can be grouped into reporting configurations. A sequence diagram showing this flow is presented below.

Note that there are a plethora of details to flesh out here, not least of which is access control. However, this document leaves all of these concerns as a future exercise. We primarily stress that *the adoption of a comprehensive measurable unit taxonomy is not a prerequisite for effective x-media measurement*.

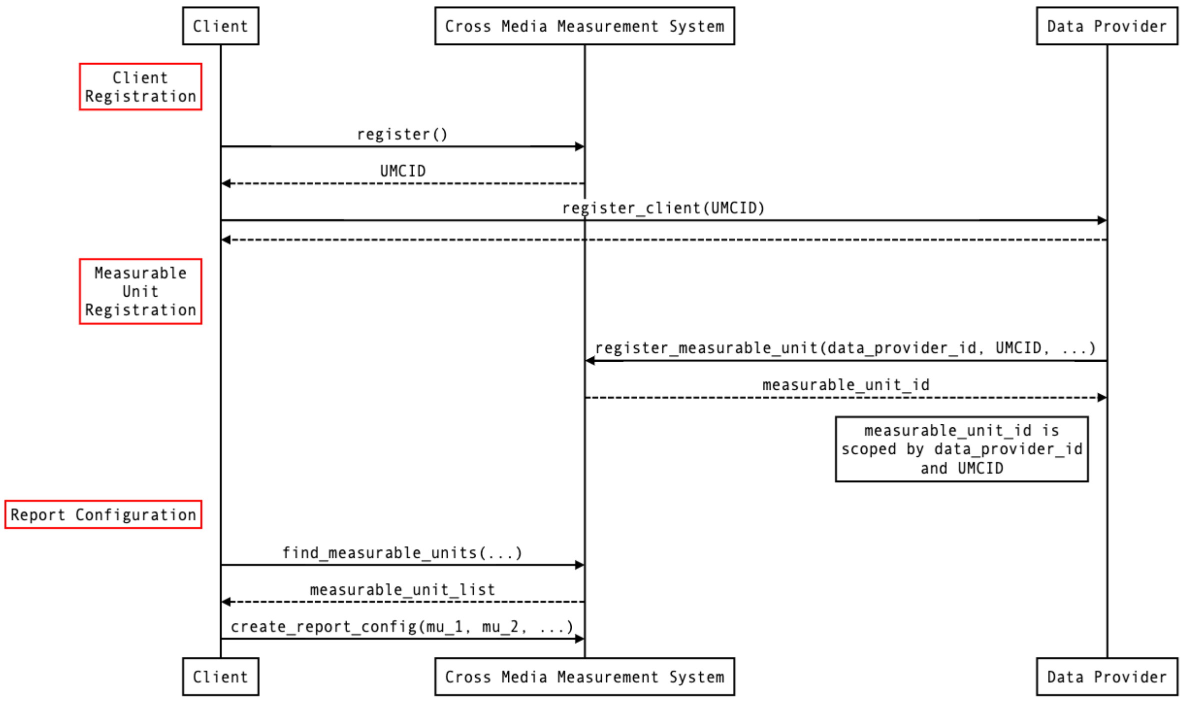
Client
Cross Media Measurement System
Data Provider
Client Registration
register()
UMCID
register_client(UMCID)
Measurable Unit Registration
register_measurable_unit(data_provider_id, UMCID, ...)
measurable_unit_id
measurable_unit_id is scoped by data_provider_id and UMCID
Report Configuration
find_measurable_units(...)
measurable_unit_list
create_report_config(mu_1, mu_2, ...)
Client
Cross Media Measurement System
Data Provider

Fig: Flow of Client Registration, Measurable Unit Registration, and Report Configuration

Client APIs

Overall this document does not delve deeply into any aspect of the Client APIs, except to note that at minimum several APIs will be required in order to support the computation of reach and frequency reports. At a minimum the following APIs or similar will be required:

- Client Registration Service
    - allows clients to sign-up
    - allows clients to delegate access and assign roles (e.g. some employees might be able to view reports, but not configure them)
    - this could easily become multiple services

- Measurable Unit Search Service
    - allows advertisers/content owners to search for their campaigns/content by name, data provider, and other criteria
- Report Configuration Service
    - allows clients to configure a reach and frequency report (henceforth just report) by specifying one or more measurable units across one or more data providers
    - allows clients to schedule the frequency of report generation for each configuration
    - allows clients to specify metadata for slicing measurement units associated with the reports (e.g. demographics, geo, etc.)
    - this could also become several services, for example it may be worth splitting up the APIs that provide for measurement unit grouping and the APIs that provide for report scheduling
- Reporting Service
    - This will be the service for retrieving report results, and as above it is not unreasonable to think that this could expand to become several services as well

Note that the registration and report configuration flows will also require UIs or some other kind of manual tooling in order to drive them. It is possible that these could be provided by the authors of the XMMS, by third parties, or both.

From this point forward this document assumes the existence of both UMCIDs and report configurations, where the latter can be thought of as a UMCID scoped list of (data_provider_id, measurement_unit_id) tuples and a time specification indicating the time periods for which the reach and frequency report should be generated.

Data Provider APIs

The set of Data Provider APIs are as follows:

- Data Provider Registration Service
    - This could be an actual live service or manually updated configuration depending upon anticipated volume of requests.
    - It exists in order to register the existence of Data Providers and apply equally to TV and digital impression providers.
- Measurement Unit Registration Service
    - This is called by Data Providers on behalf of clients to register the existence of campaigns/content.
    - Measurement Unit registration entails
        - Providing an ID for the measurement unit, which is scoped by both the Data Provider ID and the owning UMCID.
        - Providing additional metadata (e.g. a description) to make it possible for clients to search for and find measurement units of interest.

        - Exactly how we do this for TV data is an open question, but based upon our current knowledge it appears feasible.
- Metric Requisition Service
    - This is the main service for moving measurement data between Data Providers and the Private Reach and Frequency Estimator.
    - A Metric Requisition is a structure that represents a sketch that must be generated by a particular Data Provider on behalf of a particular client.
    - A Metric Requisition includes metadata that defines what set of impressions belong in that sketch. For example it might specify one or more demographic labels, a time range, or other things. The exact details of what metadata can be specified will be determined in conjunction with MRC guidelines.
    - This *pull model* is required since it is the responsibility of the Data Provider to aggregate data before sending it, and because the filtering criteria may change based upon client input.
    - Note that for reports with predictable scheduling it would be possible to generate requisitions well in advance of needing the report, or even provide a single requisition that would indicate a particular sketch be sent on a regular schedule over a period of time or in perpetuity. In general we will not get into this level of detail here.
    - This service provides:
        - An endpoint to get the set of of sketches that must be generated in order to fulfill pending reports
        - An endpoint to fulfill the requisition, which includes uploading a sketch
        - It is also possible for a Data Provider to deny a requisition, for example because the sketch would not include enough users
- Publisher Data Service
    - The Publisher Data Service is a single service that wraps each of the above services and provides a Data Provider a single integration point for both digital publishers and TV into the Private Reach and Frequency Estimator.
    - While it may appear superfluous at the present time, its true purpose will become apparent once the components of the MPC protocol have been introduced below.
    - Despite using Data Provider consistently in other places we have adopted the term Publisher Data Service here. It could as well have been called Data Provider Data Service. We could also consider renaming to Impression Provider Data Service.
    - This service applies equally to both TV and digital impression providers.

In addition to the above, Data Providers will need to create services or UIs capable of receiving a UMCID from clients or their agents, but these details are out of scope here. Suffice to say that the details of this should be determined by each Data Provider in relation to their existing suite of products, though it may be possible to create some common infrastructure to make it easier.

Expanding our previous diagram to take account of both the Client and Data Provider APIs we have the following.

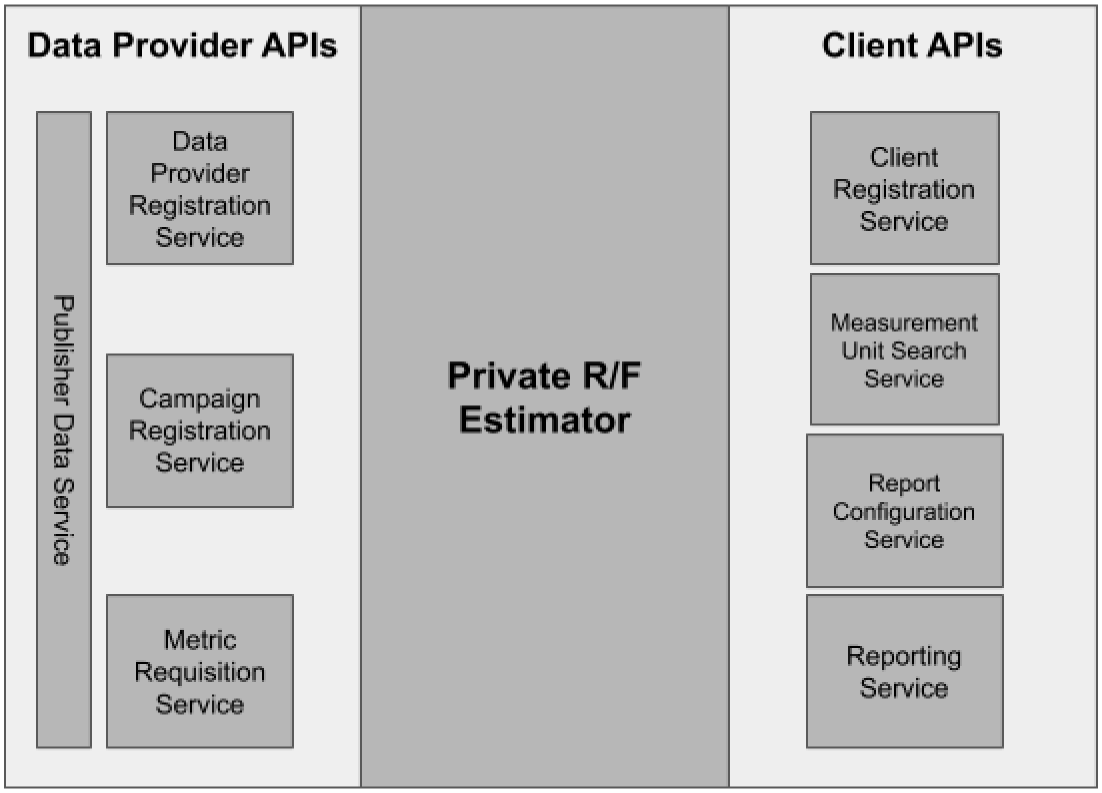
Data Provider APIs
Publisher Data Service
Data Provider Registration Service
Campaign Registration Service
Metric Requisition Service
Private R/F Estimator
Client APIs
Client Registration Service
Measurement Unit Search Service
Report Configuration Service
Reporting Service

Fig: Private R/F Estimator Core APIs

Report Scheduling

Having identified the APIs for how a client can configure a report and the APIs for how Data Providers can provide the constituent inputs of reports (i.e. sketches), the next step is to describe how these pieces work together to schedule the generation of the actual reach and frequency reports. We assume that the client has already registered their UMCID with the relevant data providers, that each data provider has registered the relevant measurement units, and that a report configuration has been created by the client.

From here the Report Scheduler will periodically inspect the set of report configurations to determine if any are due. If the report is due then the Report Scheduler will inspect the set of measurement units associated with it and determine what input sketches are required to fulfill the report. Next the Report Scheduler will look to see if these sketches are already available. If any sketch is not available then the Report Scheduler will create a Metric Requisition in an unfulfilled state for it.

As the Report Scheduler is running, Data Providers will also be polling the system for unfulfilled Metric Requisitions. Therefore, at some point after the Requisitions have been created by the Report Scheduler the Data Provider will become aware of them, subsequently create the appropriate sketches, and upload them via the Publisher Data Service. Once the sketches are uploaded the Metric Requisitions will be marked as fulfilled. We chose to have the Data Providers poll because doing so makes Data Provider integrations simpler since setting up and maintaining a service to accept incoming requests is substantially more difficult than polling, and it also opens up an attack surface.

The Report Scheduler will then notice that all of the requisitions required to fulfill a particular report are available and the report is ready to be run. The following sequence diagram shows the flow described above. Note that the method names are illustrative only and that the low level design may differ on some of the fine details.

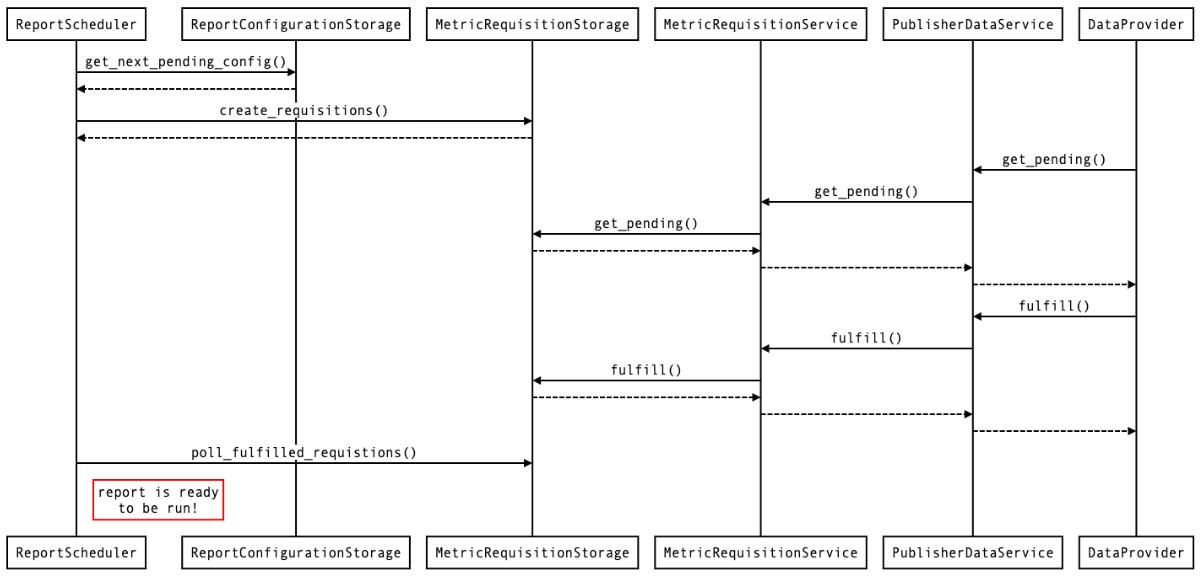

Fig: Report Scheduling Flow

Toward Decentralized Computation: Kingdom and Duchy

So far there has been a tacit assumption that reports are composed of exactly one reach and frequency calculation, however this need not be the case. Indeed, in the long-run things may actually become far more complicated, and it is likely that a one-to-one correspondence between reports and reach and frequency calculations will not hold. For example, it is easy to imagine that a single report contains reach and frequency computations corresponding to several demographic slices. Therefore it makes sense to separate the concept of the report from the pieces that make up the report, which we call Results. Moreover, it is also useful to separate the concept of a Result from the instructions required to compute them, which will call Computations.

Whereas report scheduling is a centralized process, the generation of results via a computation is a decentralized process carried out via an MPC protocol. This necessarily requires multiple independent systems communicating and working together to produce the agreed upon result. The protocol described in Privacy Preserving Secure Cardinality and Frequency Estimation requires at least two such systems, however we typically assume there will be three, which is done to help ensure that the software generalizes well and avoids making any simplifying assumptions that may arise from implementing the protocol across only two systems.

The systems that implement the MPC protocol are each made up of several components and are fairly complicated in their own right, and makes the terminology for discussing these systems rather difficult. Words like “task”, “job”, “process”, “node”, and “worker” are hopelessly overloaded. Moreover, complicated noun phrases are cumbersome, and the abbreviations of such phrases tend to be difficult to say and remember.

For these reasons we have developed a rather non-standard, but intuitive, terminology for discussing these systems. We call the centralized system responsible for report scheduling the Kingdom, while the systems responsible for running the MPC protocol are called Duchies. The following diagram illustrates how the set of APIs and components discussed above distribute over the Kingdom and the Duchies.

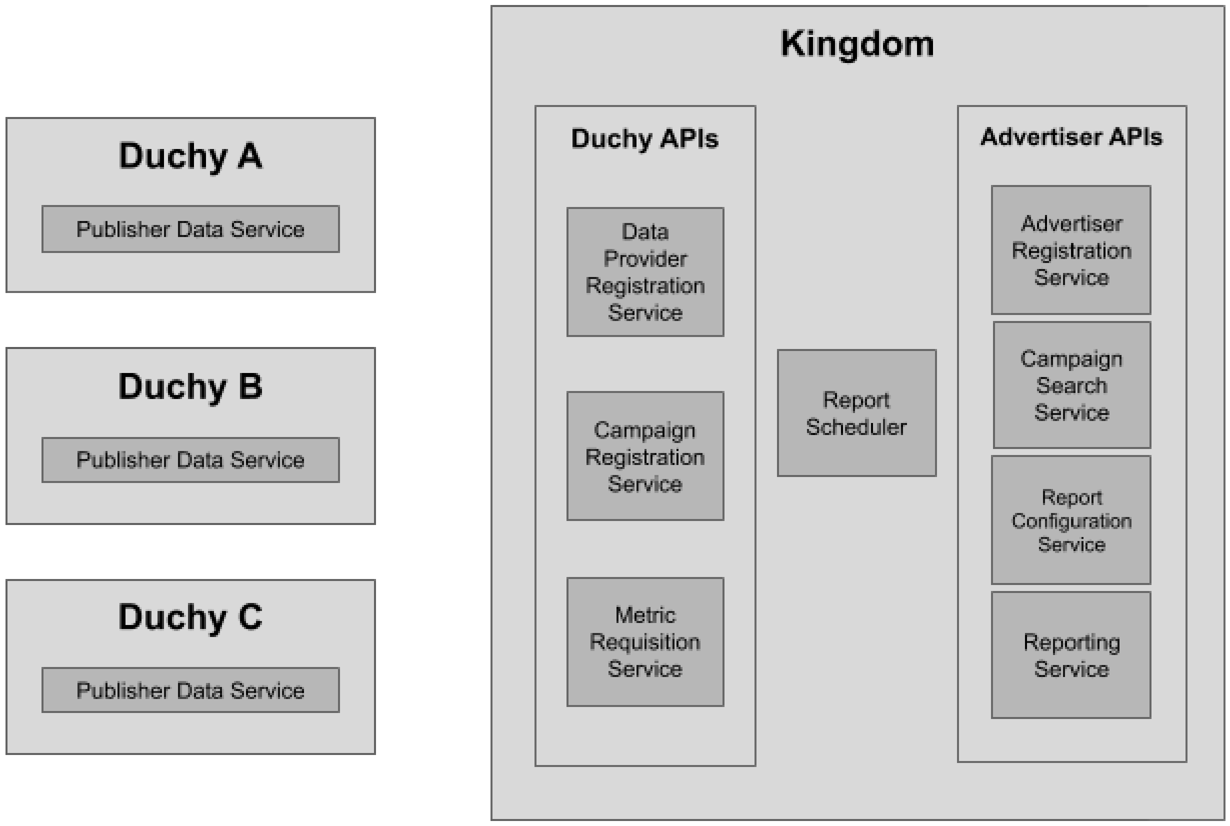

Fig: Core APIs Spread Across Kingdom and Duchies

Observe that most of the components discussed so far reside in the Kingdom, the exception being the Publisher Data Service. The justification for placing most of the APIs in the Kingdom is straightforward. There must be a central place to coordinate the scheduling of reports, which requires reconciling inputs provided by both clients and data providers.

What is perhaps controversial is placing the Publisher Data Service in the Duchy, however there are two factors for this, which when taken together make this a pretty easy decision. The first factor is the desire to simplify Data Provider integrations, which means that the Data Provider should ideally only talk to a single system, in this case either the Kingdom or a single Duchy.

The second factor was that of minimizing communication overhead, which has two considerations. The first is that encrypted sketches are reasonably large (~50MB on average) and numerous. Assuming 20 slices per measurement unit, which is a conservative estimate, means that there would be 1GB of sketch data generated per measurement unit per data provider per day. The second consideration is that the Kingdom has absolutely no use for the raw encrypted sketch data.

Taken together it made sense to give Data Providers a single system to integrate with and to make that system the Duchy of their choice.

Life of a Computation

As discussed above the Kingdom is responsible for determining when a report should be scheduled, while the Duchies are responsible for computing the contents of the report. This section considers how the Duchies determine when to undertake such computations, how a computation is carried out, and how the result of a computation is returned to the Kingdom.

To discover the existence of new computations Duchies maintain an open streaming RPC request with the Kingdom's Global Computation Service. The component of the Duchy responsible for this connection is called the Herald, which simply gathers the set of computations that the Kingdom has prescribed and then registers them with that Duchy's own Computation Storage Service. Note that each Duchy does this independently, so that all Duchies have a record of all computations. The diagram below shows this graphically. Note that components in red belong to a particular Duchy whereas the blue components belong to the Kingdom.

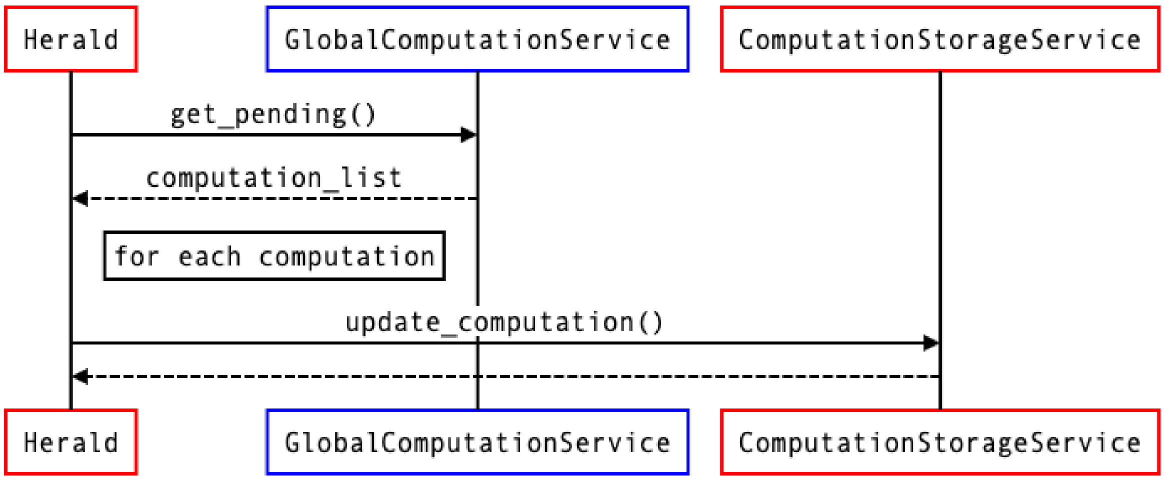

Fig: Polling for Computations

In order to carry out a computation a single Duchy must be identified as the primary. This primary Duchy is responsible for collating the sketches required for the computation from all duchies, initiating the MPC protocol, and transmitting the result of it back to the Kingdom. There are several possible ways of identifying the primary Duchy, but for our purposes here we will assume that the Kingdom simply makes this assignment. However, note that the assignment is done on a per computation basis, which allows the computational load to be spread out, since the primary Duchy is responsible for considerably more work.

The protocol also requires that the Duchies communicate in a ring. We ensure this by fixing the order of the Duchies regardless of which one is primary, and for ease of implementation the order is consistent for all computations. For ease of exposition going forward we will assume that Duchy A is the primary and that the order of communication proceeds from Duchy A, to Duchy B, to Duchy C, and finally back to Duchy A again.

While the protocol is being executed Duchies communicate by pushing messages to the next Duchy in the ring, and for the purpose of receiving these messages, each Duchy provides a Computation Control Service. Cryptographic operations required at each round of the MPC protocol are both long running and CPU intensive, so upon receiving messages the Computation Control Service enqueues work for offline processing via the Computation Storage Service before returning to the caller. The Computation Control Service does not do any processing of its own. Instead the actual processing is handled by another component called the Mill, which is a daemon that is responsible for processing enqueued work. The Mill is the component in which all of the crypto operations take place, and it is also responsible for pushing messages to the downstream Duchy's Computation Control Service.

Thus the part of a Duchy responsible for running the MPC protocol is composed of a Herald, a Computation Storage Service, a Computation Control Service, and a Mill. The dependencies between these components across two Duchies are shown below. Also note that the Mill should be seen as a pool of jobs, though for our purposes here that detail is not too important.

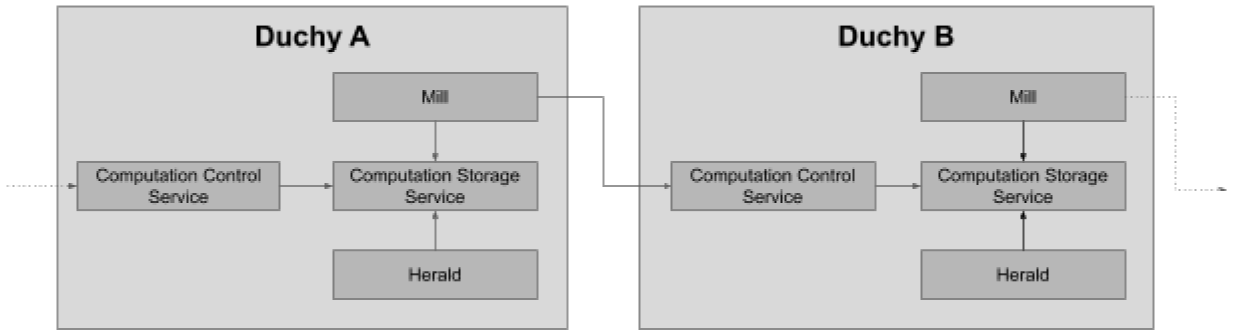
Duchy A
Mill
Computation Control Service
Computation Storage Service
Herald
Duchy B
Mill
Computation Control Service
Computation Storage Service
Herald

Fig: Duchy Component Dependency Diagram

The following sequence diagram shows the flow of data in a single Duchy and how adjacent Duchies communicate with one another. Duchy A is shown in Red, Duchy B in green, and Duchy C in blue.

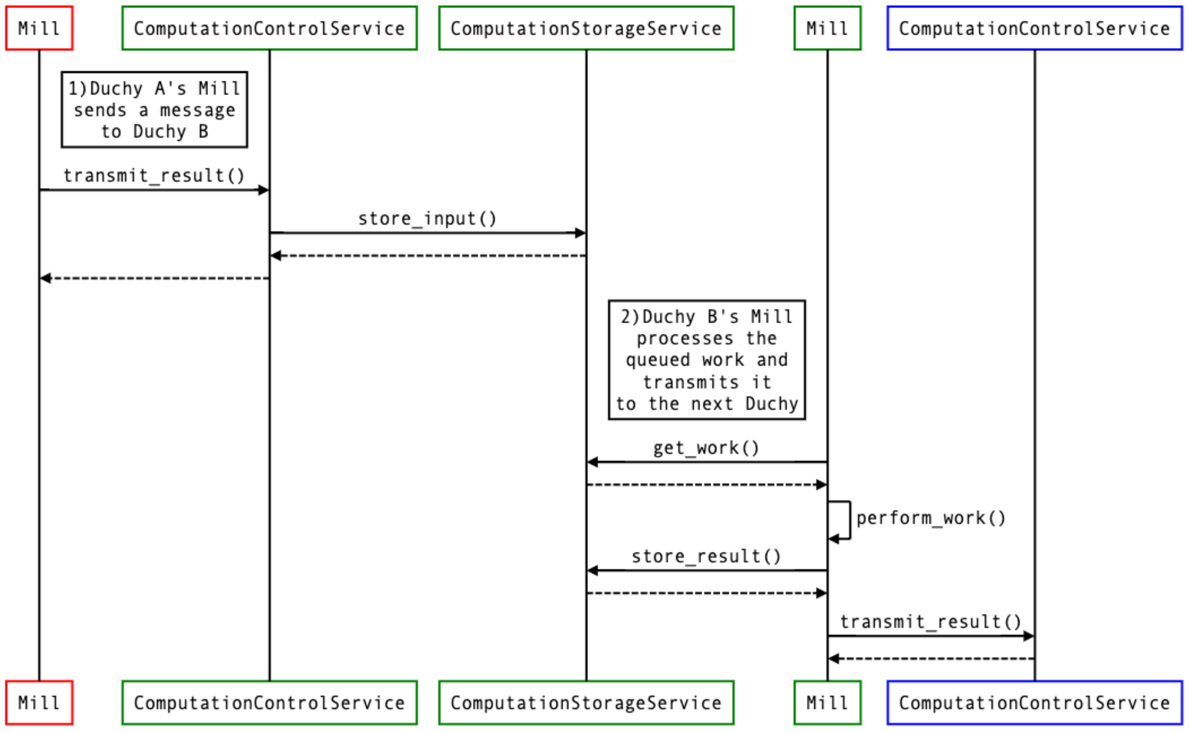
Mill
ComputationControlService
ComputationStorageService
Mill
ComputationControlService
1)Duchy A's Mill sends a message to Duchy B
transmit_result()
store_input()
2)Duchy B's Mill processes the queued work and transmits it to the next Duchy
get_work()
perform_work()
store_result()
transmit_result()
Mill
ComputationControlService
ComputationStorageService
Mill
ComputationControlService

Fig: Duchy Data Flow

From here it is easy to understand the execution of the entire protocol across all Duchies. Recall from [1] that the protocol requires two rounds of communication, that the first round is responsible for blinding register IDs, and that the second round is responsible for computing reach and the frequency distribution. Moreover, recall that each party in the protocol (aka Duchy) must apply its key to each register and/or register value in order for the protocol to faithfully compute the result. The execution of the protocol is shown in the following sequence diagram. Note that the diagram assumes that all of the Duchies have already polled the computation from the Kingdom. The internal flow of each Duchy is omitted, but it is as shown above in the Duchy Data Flow diagram.

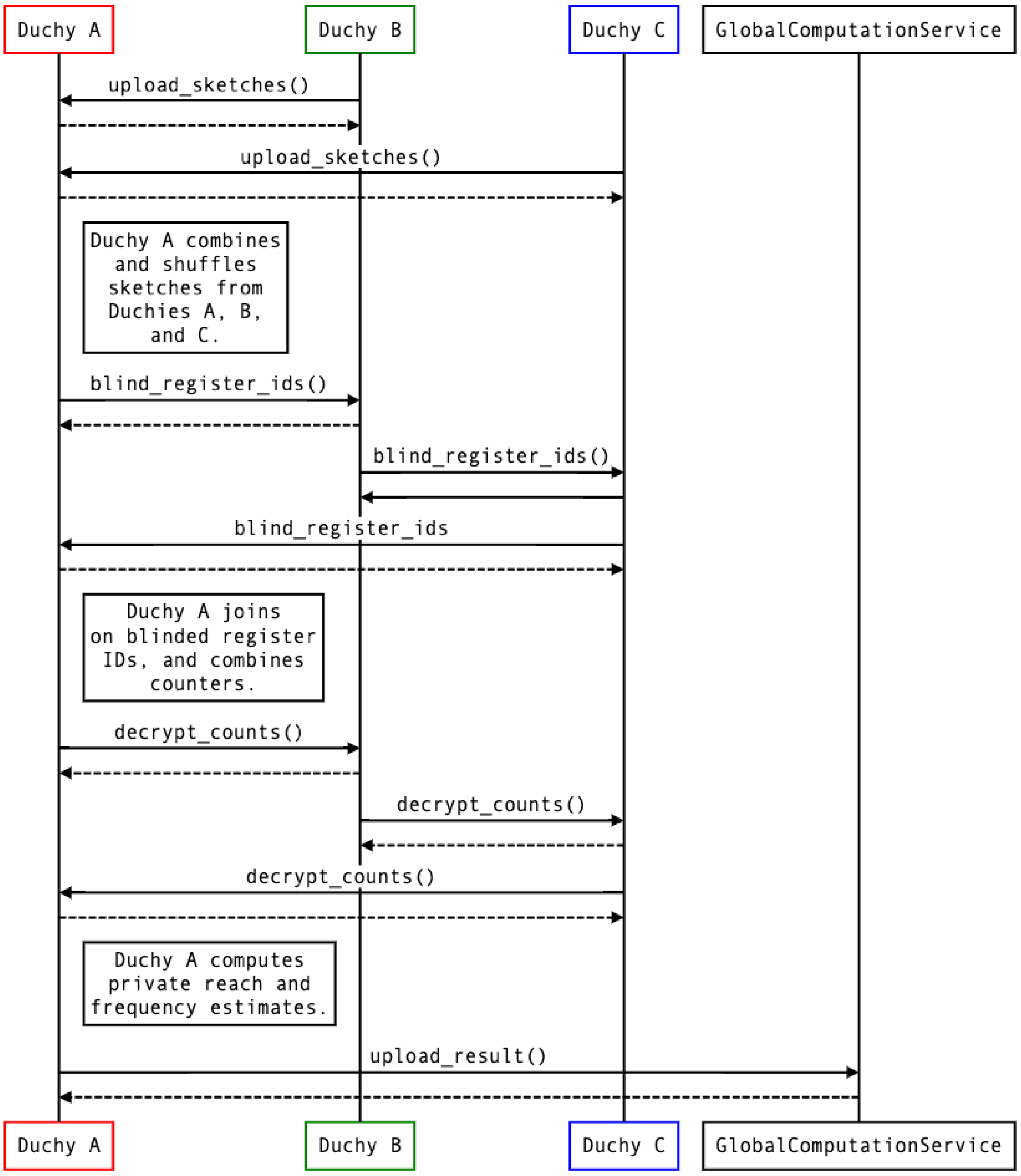

Fig: Protocol Flow

Finally, once the result of the computation is delivered to the Kingdom the report can be updated with it and the report can be made available via the Kingdom's Client APIs.

A full inter-system dependency diagram is shown below.

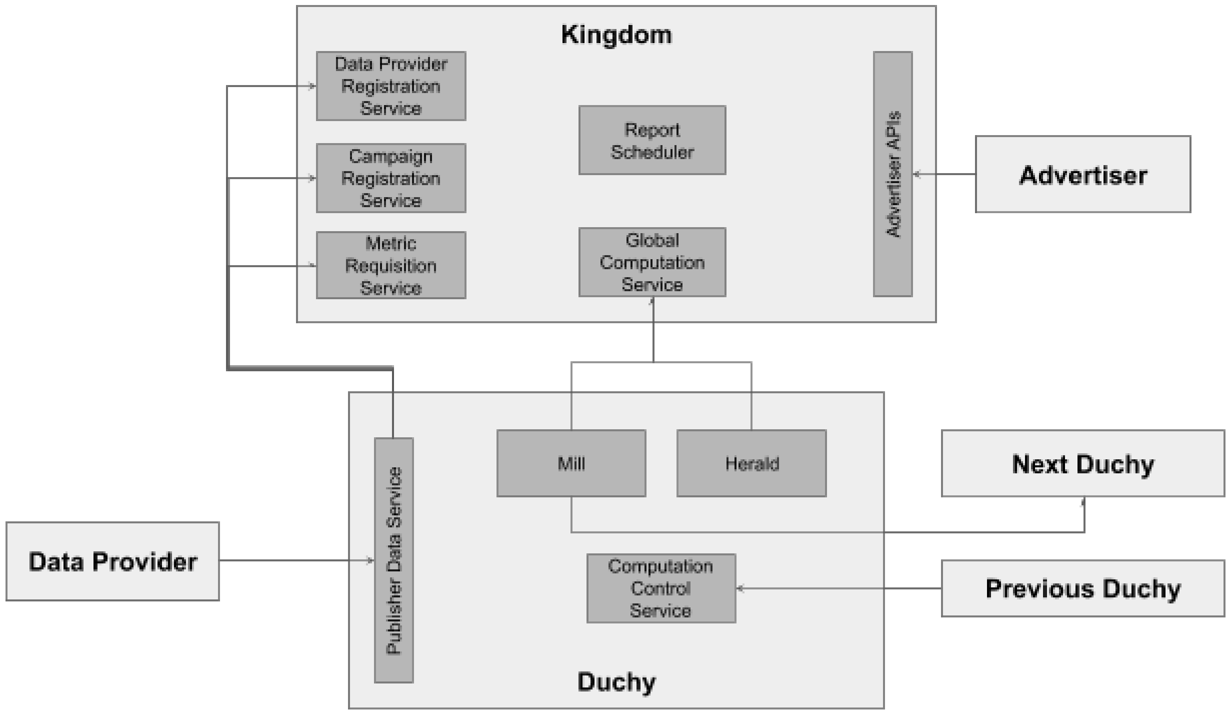
Kingdom
Data Provider Registration Service
Campaign Registration Service
Metric Requisition Service
Report Scheduler
Global Computation Service
Advertiser APIs
Advertiser
Publisher Data Service
Mill
Herald
Next Duchy
Data Provider
Computation Control Service
Previous Duchy
Duchy

Fig: Inter-System Dependency Diagram

Infrastructure

Tech Stack

We use Kotlin for building all parts of the Private Reach and Frequency Estimator that are not compute-intensive and that do not require extensive performance optimization, which essentially comes down to all of the parts that are not involved in cryptographic workloads. This includes implementing handlers for RPC services, interacting with databases, and managing work queues.

Kotlin provides static typing, generics, a compact syntax, and higher-level abstractions than C++ and Java, which together we have found makes it a rather productive language. Better memory management avoids a whole class of security vulnerabilities related to lack of memory safety in C++. Kotlin's type system also enforces explicit handling on the null case for nullable types. This means that Kotlin does not suffer from NPEs like Java or undefined behavior that results from dereferencing NULL pointers in C++. Coroutines make writing asynchronous code simpler and less error prone than in C++ and Java. Finally, targeting the JVM makes it easy to take dependencies on Java code in Kotlin projects.

We use C++ for compute-intensive algorithms like ElGamal encryption and computing and aggregating sketches. The core of our MPC computation is built using the Private Join and Compute library for C++, which is itself built upon the OpenSSL library. We extend it with several primitives related to our application and wrap the resulting C++ code with JNI and call it from Kotlin. This forms the basis of the Mill.

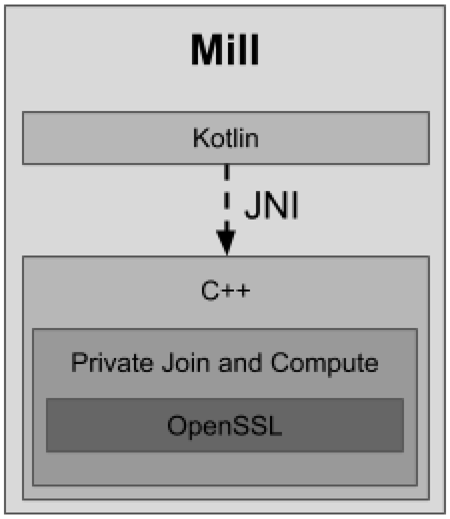

Fig: Mill Tech Stack

We use gRPC as our RPC framework. gRPC is easy to use and since it uses HTTP/2 it is much more efficient than HTTP REST APIs for transmitting large amounts of data. gRPC has native support for Kotlin as well as many other mainstream programming languages including Java and C++. We plan to write all gRPC clients and services for Private Reach and Frequency Measurement in Kotlin.

We build JVM binaries using Kotlin and package these in Docker images. Docker containerization allows us to standardize the execution environment avoiding issues with operating system versions and other potential sources of incompatibility. We currently target distroless/java, the default configuration for rules_docker. Distroless images are lightweight and minimize threat surface area by eliminating unnecessary components. For example there is no shell in the distroless/java image. Distroless supports debug images for development that do include such niceties as shells.

We use Kubernetes to deploy, manage, and scale our containerized applications. The combination of Docker and Kubernetes is very common and quite powerful. We plan to build Docker images and push them to a public image repository. Duchy and Kingdom operators may deploy these pre-built and tested images to their QA and production environments without needing to maintain their own build environments. Containerization eliminates the need to maintain compatible execution environments. Kubernetes provides a standardized platform, processes and object model for deployments. In addition we will provide Kubernetes configuration templates for operators to customize. This will allow us to easily scale up the number of operators as necessary. Furthermore Kubernetes is available as a service from major cloud providers including AWS, Azure and GCP and it is also possible for an organization to set up and maintain their own Kubernetes cluster. This provides a cloud-agnostic solution for handling Kingdom and Duchy deployments.

For storage we require a scalable relational database and object storage. For Google Cloud Platform, we are currently targeting Google Cloud Spanner and Google Cloud Storage but we also provide an abstraction layer for each to facilitate implementing adapters for different storage systems.

Developer Toolchain

We use Git for version control and will release the project in a public GitHub repository. Git and GitHub are de facto industry standards for open source projects.

We use JUnit4, Mockito and Truth for writing tests in Kotlin. For C++ we use Google Test.

We use Bazel for our build system. Bazel supports the multi-language use case well. Bazel caching also speeds up builds and Bazel is easy to extend. There are Bazel rules for a number of tools we use such as rules_docker for building Docker images.

Our builds target Linux. We develop on Debian but none of our dependencies should be specific to a particular Linux distribution.

For configuration we are leaning towards CUE. Managing Kubernetes configuration as separate YAML files across production, QA, dev and local environments rapidly becomes unwieldy. CUE supports schemas and types and requires minimal boilerplate. There are CUE export rules available for Bazel.

We use IntelliJ and CLion for our IDEs, though of course, it is possible to use any editor. IntelliJ supports Kotlin and Git well. There is also a good Bazel plugin. CLion meets our C++ needs.

In addition to the components described above, we adamantly support automated unit and integration testing both locally, before submits, and as part of a continuous integration and deployment workflow. To this end, for local development and testing, we are leaning towards Kind (Kubernetes-in-Docker) as a lightweight and relatively easy way to create and deploy to local Kubernetes clusters. For continuous integration (CI) we are leaning towards Prow. Prow is the CI/CD system of the Kubernetes project. Prow allows you to run container images in response to GitHub webhooks. We plan to use a Bazel image for building and testing. We are leaning towards Prow for continuous deployment (CD) as well but do not have a clear strategy in mind for CD yet.

Performance Testing

In addition to building a working Proof of Concept we also intend to run performance tests of the MPC-oriented parts of the system, which means measuring the system's performance after an R/F report computation has been scheduled by the Kingdom. We are primarily interested in CPU usage associated with the entire suite of cryptographic operations, network usage, and wall clock time for computing an R/F report. Secondarily we are interested in measuring database usage and interaction, RPC latency, RPC errors (especially timeouts), and a number of other things.

A tertiary goal of this performance test is to provide data to help determine the costs associated with operating a duchy, however details of how the costs might be estimated, and specifically which data are required to support costing, are out of scope here.

Specifically we plan to measure the following:

- Wall clock time from computation begin to computation end from kingdom's perspective
- CPU time per computation stage per duchy
- CPU time for each sketch-level crypto operation
- Cross duchy-bytes transferred per duchy per stage
- RPC latency per-duchy for all x-duchy communication
- RPC errors (esp. timeouts) per duchy for x-duchy communication
- Bytes of storage used (RDBMS and blob; intermediate results as well as outputs)
- Bytes of network egress from storage consumed
- Count of database read/write operations

The first four of these are the highest priority.

Each of the above metrics will be analyzed in order to understand their relationship to the number of input sketches and the *combined* sketch size measured in buckets. To begin with we will vary the number of data providers across the following range (10, 50, 100). Reach can be the same per publisher to begin with and will be varied along the following range (10,000, 1,000,000, 10,000,000). Universe sizes should be adjusted to ensure sufficient overlap. To generate the data we will use Reach Scenario 1 from the evaluation framework.

To account for variance in single measurements, each data point will be evaluated 10 times. This means that we will carry out 90 computations (3*3*10). Since most metrics are per-duchy, this will give us 20 samples for many operations (i.e. non-primary duchy work), 30 for some, and only 10 for others. Understanding overall system resource usage in the case where there are more than three Duchies will be accounted for by extrapolating from the performance of the two secondary Duchies. This makes sense because secondary Duchies all do the exact same computations.

Note that varying per publisher frequency should not impact scaling as the number of sketch buckets is not impacted, but we may want to vary frequency anyway. If so, frequency scenario 1 from the evaluation framework will be used for this purpose.

Next Steps and Future Directions

To reiterate, the purpose of designing and building this system in its current form was to test the feasibility of an MPC-based approach to reach and frequency estimation at scale. We also plan to open source the code under the Apache 2.0 license once we run a complete end-to-end correctness test of the system described above.

Additional documentation describing the following components in depth may also be created.

- Publisher Data Service
- Report Scheduling
- Decentralized Computation of Results and Cryptographic Operations
- Production and Test Infrastructure
- Data Provider Integration Guide and SLOs

These would be written after the code has been made available.

Beyond thinking a bit about UMCID, we have done essentially zero work on Client APIs. Rather we expect this to be the subject of an external workstream that will hopefully commence soon. It is expected that decisions about the Client APIs could impact the work described above.

Finally, we believe that the framework that we have built for computing reach and frequency estimates should be readily extendible to similar MPC protocols. In particular we believe that it is readily adaptable to computing Secure Universal Measurement IDs (SUMIDs) and several flavors of Multi-touch Attribution (MTA).

References

[1] Privacy Preserving Secure Reach and Frequency Estimation
https://research.google/pubs/pub49177/