Implementing Distributed Query Execution in Privacy Backplane

Anna Cai
Northwestern University
Evanston, IL, USA

Bowen Cheng
Northwestern University
Evanston, IL, USA

Blake Hu
Northwestern University
Evanston, IL, USA

Lianhao Zheng
Northwestern University
Evanston, IL, USA

ABSTRACT

The Privacy Backplane Distributed System (PBDS) provides a framework for enforcing user-defined privacy policies over sensor data in real time. In this work, we focus on the design and implementation of PBDS's core query engine: a distributed, DAG-based runtime that orchestrates the deployment and execution of data-flow operates across peer-to-peer nodes. Furthermore, we detail the mechanisms for query initialization and execution, localization, and group membership management via the gossip protocol. Beyond the implementation itself, we discuss the architectural choices that underpin PBDS such as peer discovery with libp2p, definitions for user, analyst, and venue nodes, and the use of Rust. This implementation lays the groundwork for future performance evaluation and integration of advanced policy and localization features.

1 INTRODUCTION

Ubiquitous sensing is now deeply embedded in modern environments, from retail venues to public campuses, introducing the urgent need to protect individual privacy in real time. Cameras, microphones, Bluetooth beacons, and other ambient sensors generate continuous streams of sensitive personal information like location, identity, and behavior, often without meaningful consent or visibility into how the data are being processed. Existing legal frameworks such as the General Data Protection Regulation (GDPR) and the California Consumer Privacy Act (CCPA) offer broad protections, but often lack the fine-grained enforcement mechanisms required to operate effectively in real-time settings.

The Privacy Backplane addresses this gap by proposing a legal-technical framework in which users define and control the policies that govern how data about them may be accessed, transformed, and exported. Users specify policies–Require, Forbid, or Don't Care–on named data attributes that reflect their personal expectations. These policies are embedded in the Privacy Backplane Distributed System (PBDS), a peer-to-peer system that enforces them at the point of data collection and processing, without assuming trust in venue operators or relying on a central server.

This decentralized, policy-aware architecture supports more nuanced interactions between users and environments. Users may have different sensitivities and venues have legitimate operational needs. PBDS offers a flexible alternative to binary opt-in/opt-out mechanisms. Users retain control over how they are sensed, and venues are permitted access only to the extent allowed by policy.

In this paper, we present our contributions to the PBDS prototype:

- A distributed DAG-based query runtime
- A modular localization mechanism that embeds (x, y, z) user tracking directly into UWB sensor nodes, enabling each venue node to receive relevant user locations in an uncoordinated way.
- A gossip-based peer discovery and group membership protocol that supports deployment of query pipelines across heterogeneous nodes.

2 RELATED WORK

There have been multiple approaches to protecting personal privacy, both through regulatory-based approaches and technical approaches.

2.1 Regulatory Frameworks

An example of a privacy protecting regulatory framework is the European GDPR, which places restrictions on how personal data is collected, used, and transferred. Another example would be U.S., California's CCPA, which seeks to provide transparency and control to individuals over the collection and use of personal data. In addition, Texas and Illinois have regulatory frameworks in place to protect citizens from the collection and use of biometric data. While these regulations embed privacy protections into law, there are questions to their effectiveness and enforceability. A fundamental limitation is that regulatory frameworks rely

heavily on self-reported documentation and the threat of investigatory actions.

2.2 Technical Approaches

There has not yet been a holistically designed approach that ensurs trused operation at each layer of the software stack, all the way down to data captured by embedded systems. That is the goal of the Privacy Backplane. Our vision is to use regulation to support a technical system capable of negotiating and enforcing policies acceptable to operators and users, as opposed to specific policies.

3 THREAT MODEL AND REQUIREMENTS

3.1 Threat Model

Our primary security goal is to ensure that all data collection and access occur strictly within the boundaries defined by user nodes and venue nodes. We assume the presence of a potentially malicious analyst with full control over the sensing, processing, and storage infrastructure, including the OS and hypervisor. The attacker may tamper with, delete, reorder, or replay network traffic within the system. Our focus is on preserving the integrity of user policies, ensuring correct binding between policies and sensed data, and maintaining the confidentiality of sensed data. We do not address malicious users or availability attacks.

3.2 System Requirements

To achieve these objects, every node type (User, Venue, Analyst) should run on a hardware TEE such as ARM Trustzone or Intel SGX to provide attestation, sealing, and memory protection. We do not explicitly consider side-channel attacks or implementation flaws.

4 POLICY AND QUERY MODEL

Our policy framework expresses venue and user requirements as triples of the form *Require*, *Forbid*, or $Don't-Care$ over named data attributes. A *Require* rule stipulates that a particular attribute (for example, precise GPS coordinates or raw video frames) must be available to downstream operators. A *Forbid* rule prohibits any exposure of that attribute beyond the local operator. A $Don't-Care$ rule indicates no specific constraint. At query setup time each pair of adjacent nodes exchanges their policy triples and determines compatibility by checking that no attribute is simultaneously *Require* and *Forbid*. Common $Don't-Care$ cases are handled via a fast-path rule check; when conflicts arise between user and venue nodes, the nodes engage in a more detailed negotiation protocol that considers attribute hierarchies and redaction capabilities. Once compatibility is established the resulting effective policy is enforced with the TEE during execution.

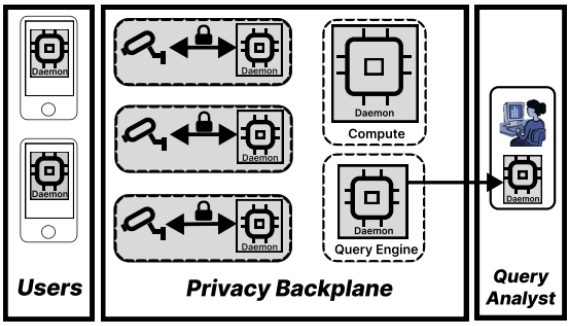

Figure 1: Privacy Backplane architecture [1]

Queries are represented as directed acyclic graphs whose vertices represent data-flow operators. Each operator declares the attributes it consumes and produces, and the backplane runtime verifies at instantiation that the operator's input requirements satisfy the user's effective policy. An operator can either input data, transform data, or output data. The runtime enforces that no data leaves a node unless the node's policy check has passed and the operator's output exactly matches the declared attribute set.

5 SYSTEM ARCHITECTURE

5.1 Networking

The architecture of the Privacy Backplane is that of a distributed trusted execution environment (Distributed TEE, or DTEE) that extends from IoT sensors and the edge to processing and storage infrastructure in the cloud. Each DTEE component uses hardware and software TEE features to handle infromation in a safe and secure manner. The DTEE components interoperate as a distributed system to ensure that data privacy, security, and integrity are maintained from the time information is captured to when it leaves the DTEE with all negotiated polices having been aplied, or until deletion from the DTEE. To form the distributed system, each component has a Privacy Backplane Daemon (PBD) running within a hardware TEE. Together, PBDs form the Privacy Backplane Trusted Overlay Network (PBTON), a peer-to-peer overlay that provides authenticated, encrypted channels via mutual attestation. Sensor data is ingested into the local PBD which hosts two core engines: a Policy Engine that reconciles user and venue policies and a Query Engine that executes data-processing DAGs in a sandboxed WASM runtime.

Throughout the implementation of the PBDS, many design choices have been made. The first of which is to design the Privacy Backplane as a distributed peer-to-peer system rather than using a client-server architecture, strengthening

availability, privacy, and scalability. By distributing networking and computation across every PBD, we eliminate the single point of failure that would cripple the system if a central server went down, instead routing queries dynamically around offline or compromised nodes. This decentralization also keeps raw sensor data and enforcement of user-defined privacy policies local to each hardware TEE, so sensitive information never traverses a central hub. As venue nodes and sensors grow, the overlay balances the load without costly central servers, and edge-side DAG transforms execute with minimal latency. Trust is managed through mutual attestation between peers, aligning with a threat model that assumes any node might be attacked and must be isolated rapidly, while intermittent or partitioned networks can continue operating independently and reconcile state when connectivity returns. Finally, onboarding new devices and sensors is as simple as gossip-based peer discovery. Overall, leveraging existing node resources makes this design more cost-effective than building out dedicated centralized infrastructure.

The PBD is built on top of libp2p, an open source networking library. Libp2p was chosen for its solution to global scale peer-to-peer networking. Implementing networking stack features like reliable peer discovery or encrypted channel negotiation from scratch would divert efforts away from the core privacy and policy challenges. Lip2p provides modules for identifying peers, maintaining a distributed hash table for routing, and selecting optimal transports such as TCP, QUIC, or Websockets. By delegating these low-level networking concerns to libp2p, the focus can be on building and TEE-backed daemons, designing privacy-preserving DAG operations, and refining policy negotiation logic. PBD is implemented in Rust. This is because the two largest implementations for libp2p are in Go and Rust. While Go provides robust concurrency primitives like goroutines and channels, it leaves many design and safety concerns, such as avoiding data races and managing memory lifetimes, largely up to the programmer. In contrast, Rust's ownership and borrowing model helps catch these issues early by enforcing memory and thread safety at compile time, eliminating common data races and dangling pointer bugs without imposing the runtime overhead of a garbage collector.

5.2 Node Types

There are three types of nodes with distinct roles: User, Venue and Analyst. All node types run as Privacy Backplane daemons, though the node type determines what operations are possible on the node. The User node is intended to be run on a mobile device belonging to a person seeking to enter a venue running Privacy Backplane. The User should have selected a set of privacy policies which are then communicated to a Venue node.

There are at least four different subtypes of Venue nodes, though more may be possible. A Localizer venue node determines the spatial location of a User node using an Ultra-Wideband (UWB) radio. Localizer nodes will likely also receive the User's privacy policy and associate it with its corresponding spatial location. A Data Producer venue node records or produces sensitive user data, such as by recording a video feed of the venue. A Transform venue node performs arbitrary computations on user data in a query, such as by blurring users in a video feed. An Exit venue node transfer user data out of the system, such as by uploading the video feed to the cloud, and ensures all privacy constraints have been satisfied prior to data leaving the venue nodes layer.

The last type of node, the Analyst node, is used by system administrators to make modifications to the system. Analysts temporarily attach to the system, and make modifications such as by instantiating or stopping queries. Such modifications can only be made if there are no users in the venue currently to ensure that user data does not get incorrectly processed by partially instantiated queries. Analysts are the only nodes allowed to make modifications to query state and system configuration, but do not themselves participate in queries. After modifications are made, Analysts detach from the Privacy Backplane system.

Notably, we made a distinction between Analyst and Venue nodes because they have significantly different roles and life cycles. Distinguishing between these two different types of nodes allows us to assign different capabilities and makes implementation simpler.

5.3 Queries on venue nodes

The Privacy Backplane Distributed System operates on a dataflow model in which queries are represented as directed acyclic graphs (DAGs) composed of operators that transform raw captured sensor data into outputs compatible with user privacy policies. These DAGs provide abstraction on expressing computations in the system and are designed to enforce negotiated privacy policies between users and venues. Each query DAG can consists of subtypes of venue nodes described above.

Localizer communicates user location to all the venue nodes and only venue nodes that require such data retain it to be used for the query. Although queries are represented as DAGs to allow for branching of data flows, many subcomponents take the form of operator trees, where a linear path of computation is performed from a source to a sink. The execution of queries is implemented by mapping virtual DAG nodes onto physical nodes deployed. These physical

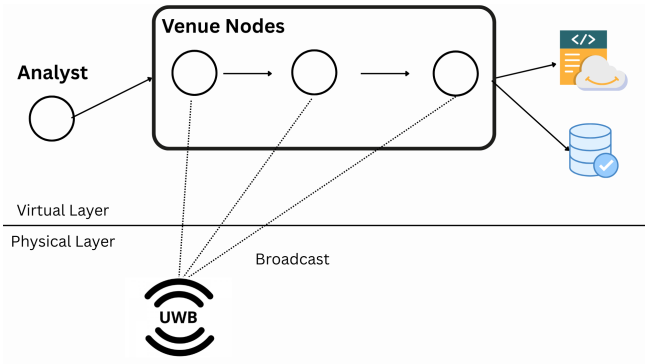

Figure 2: Example of query DAG execution

nodes, which run the Privacy Backplane Daemon, should operate within Trusted Execution Environments (TEEs) to ensure the confidentiality and integrity of all data and computation. Additionally, each Daemon includes a lightweight WebAssembly (WASM) runtime that executes sandboxed operator code within the TEE. WASM's portability and support for bounded, verifiable execution make it ideal for this distributed and security-sensitive environment, and usability for more advanced operators.

The current implementation supports query execution through an Analyst node, which generates a valid query plan by selecting appropriate operators and constructing a DAG. The Analyst node coordinates deployment by instantiating the query across all relevant virtual nodes and mapping them onto corresponding physical venue nodes. It awaits acknowledgments from each mapped node before execution via runQuery RPCs. Prior to execution, the Analyst could generate multiple alternate query plans using different combinations of operators, aiming to accommodate varying user privacy preferences. Since the exact privacy agreements of users may not be known in advance, the system should be capable of selecting or adapting a plan at runtime that complies with the policies of the users involved.

6 IMPLEMENTATION

6.1 Gossip protocol for peer discovery

To implement effective peer discovery, we wrote a simple gossip-based protocol. To bootstrap this process, all peers need to be provided the multiaddress of at least one node. On startup, the daemon dials the known peer at the provided multiaddress and exchanges network information via the Identify protocol provided by libp2p. When a peer discovers the multiaddress of a new peer, it adds the new address to its set of known addresses. Then it shares that new address with all its known peers by sending a libp2p request.

When a peer receives an address already contained in its set of known addresses, it ignores the address and does not share it with its known peers. This ensures that peer address information will eventually converge across the network and no more gossip RPC calls will be made when all peer addresses are known.

6.2 Queries

Assuming that the peers are connected using libp2p's gossip protocol for discovery and established secure communication channels, the implementation of query execution in the Privacy Backplane then begins with the Analyst node's construction of a DAG that represents the desired computation over sensor data. These queries may be preconfigured for a venue or submitted dynamically ensuing user policy negotiations. The analyst then maps each virtual operator node to an available physical node, sends InstantiateQuery RPCs, and waits for acknowledgment from the corresponding physical venue nodes. Once acknowledged, QueryRun RPCs are then sent out to all mapped nodes to begin execution of the query.

6.3 Example operators

To demonstrate the pipeline, we implemented a simple example composed of three operators, which are implemented as asynchronous tasks in Rust. The SourceGeneratorFunction acts as a data source that produces a stream of integers to simulate data input from a local sensor. This feeds into the IncrementFunction, which simply adds one to each incoming value representing a transformation step. Finally, the SinkPrintFunction receives the output and outputs it, simulating a basic form of logging or analysis. In a typical execution of this pipeline, the analyst submits the query, which is then instantiated across three venue nodes. One possible iteration would involve the source emitting the value 10, the increment function transforms it to 11, and the sink prints 11 to standard output. Despite its simplicity, this example illustrates how queries are structured and executed in a secure manner. It also provides a baseline for performance measurements, such as end-to-end query latency from the moment the analyst submits the request to when the result is received. Moreover, each function runs as an independent Daemon within a TEE, and data is passed securely between nodes using the libp2p swarm.

6.4 Context

Privacy Backplane Distribution System is centered on an asynchronous event loop running on each physical daemon that enables decentralized coordination among trusted nodes. This behavior is orchestrated in context.rs, which initializes the system and enters a multi-tasking loop to handle peer communication and query deployment. The main function

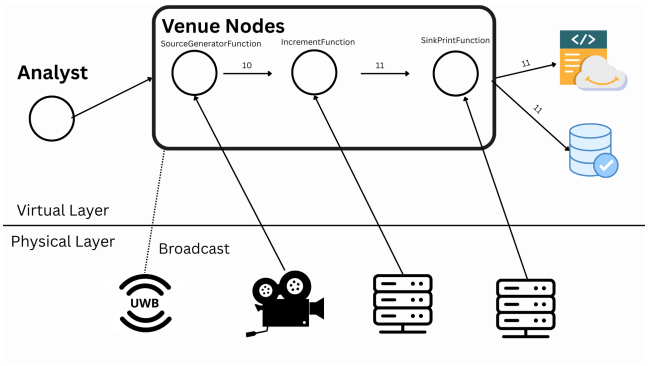

Figure 3: Example Operators execution flow

begins by parsing the runtime configuration and initializing a local Context object. This Context encapsulates the node's identity, role, networking interface, and internal state. More specifically, the node's identity is established using an Ed25519 cryptographic keypair, either loaded from a base64-encoded configuration or generated at runtime. This keypair not only provides a persistent and unique PeerId for the node but also enables secure and authenticated communication in the libp2p peer-to-peer network. Once the Context is constructed, the program concurrently spawns three tasks: running the context loop (context.run()), connecting to known peers (add_peers_from_config()), and optionally setting up a query (try_setup_query()), if provided at launch.

The core of the system lies in the asynchronous select! loop within Context::run(). This loop serves as the node's central event handler, reacting to incoming network message events. Network events are generated by libp2p, which manages peer discovery, connection maintenance, as well as instantiating and running queries.

Privacy Backplane uses the libp2p framework to implement a peer-to-peer communication stack. Specifically, it leverages libp2p's behavior system to define the network protocols each node supports. A request-response protocol is specifically used to exchange structured messages between peers, which ensures that when a query is instantiated across the network, each peer explicitly acknowledges (ACKs) the request before computation proceeds. These acknowledgments aim to provide the necessary synchronization guarantees that a query is fully deployed and ready to run with corresponding policies being actively enforced.

In particular, the event loop in Context::run() is driven by the libp2p::Swarm, which manages socket connections, handles protocol negotiation, and continuously polls for network events. It is parameterized over a NetworkBehaviour, which defines the set of supported protocols and their associated logic. In our system, this behaviour includes custom request-response handlers for query instantiation, peer information exchange, UWB localizer data propagation, and peer gossip messaging. When a network event occurs, such as an incoming request, a connection established, or a peer disconnect, it is routed through the Swarm to the appropriate handler defined in the behaviour. This enables a clean separation of concerns, where the Swarm manages the low-level I/O and multiplexing while the behaviour defines the high-level semantics of peer interaction. The Context owns the Swarm and integrates it into the select! loop, where it concurrently responds to both network events from Swarm and internal control commands. This design allows the Privacy Backplane to maintain a responsive, event-driven runtime that supports decentralized coordination and policy-aware execution of distributed queries.

6.5 Analyst

The primary purpose of an Analyst is to coordinate the instantiation of queries. To that end, the Analyst exposes a public setup_query method that performs all steps needed to instantiate a MappedQuery across the correct Venue nodes. To accomplish this, the Analyst stores MappedQuery objects and tracks the state of the query at each participating Venue node. To set up a query, the Analyst saves a copy of the MappedQuery and then sends a BeginInstantiateQuery command to the Context. The Context defines the request-response handlers for each step of query instantiation. First, the MappedQuery is sent to all participating Venue nodes, which instantiate their assigned operators and report to the Analyst that the query has been instantiated at that Venue node. After all participating Venue nodes report to the Analyst that the query has been successfully instantiated, the Analyst then sends RunQuery requests from the leaf (Exit) nodes to the root (Data Producer) nodes. After all participating nodes have successfully responded to the RunQuery request, the query is fully instantiated and running.

Analyst nodes can also issue StopQuery responses to stop and tear down a query, although this has yet to be implemented. This will likely involve stopping query operators starting from the root nodes and proceeding to leaf nodes. Once the query has been stopped, the MappedQuery may be automatically dropped when no longer referenced, as it is held in an Arc.

6.6 Localization

In earlier work, localization established the link between physical space and privacy polices by modeling users, sensors, and venues as spatial volumes defined with (x, y, z) coordinates. This approach ensure that privacy enforcement is triggered only when a user's volume intersects a sensor's point of view. Each volume is associated with a tag_id rather than a user identity, preserving pseudonymity.

In our current implementation of localization, rather using WiFi or GPS data, we track tag_ids through the use of UWB Sensors. The sensors stream serial output including (x, y, z) coordinates. The Privacy Backplane Daemon then encodes a sphere around the user to represent the spatial volume of the user. Venue nodes are able to gossip these tag_ids across the overlay, enabling policy negotiation without sharing raw coordinates.

7 FUTURE WORK

One key area for future development is the introduction of distributed consensus mechanisms to manage query state across the system. In the current implementation, queries are instantiated and managed locally at each node without a global coordination layer. While this is sufficient for single-venue or small-scale deployments, larger or more dynamic deployments will require stronger consistency guarantees to ensure correctness and safety of query execution.

We propose implementing a global locking mechanism to coordinate the state of all active queries within the system. This lock would ensure that concurrent modifications to query graphs do not result in conflicting states across nodes. In particular, we want to support scenarios where multiple analyst nodes may issue query-related commands simultaneously. In addition, this approach enables us to handle the case where, during a network-partition scenario, some venue nodes may remain unaware of newly imposed rules while users inside the venue continue to submit or modify queries. The global lock and accompanying partition-detection logic will ensure that no node applies changes out of order or without full visibility of the current user policies.

To implement this locking mechanism, we plan to integrate a consensus protocol, specifically the Raft algorithm, into PBDS. Raft provides a fault-tolerant, leader-based model for achieving consensus in a distributed system and has been widely adopted due to its understandability and practical performance. By using Raft to manage query state transitions, we can ensure that all nodes participating in a given query pipeline agree on its structure and policy constraints, even in the presence of failures or message delays. Integrating consensus into PBDS represents a critical step toward making PBDS a robust privacy infrastructure.

8 CONCLUSION

In this paper, we have presented the design and implementation of PBDS's core query engine, a distributed, DAG-based runtime that executes data-flow operators across peer-to-peer nodes. We have shown how queries are initialized and executed in a policy-aware manner, how group membership is managed via a lightweight gossip protocol, and how localization integrates with our runtime to associate policies with user presence. Our use of libp2p for peer discovery, the clear separation of user, analyst, and venue roles, and the choice of Rust for system components together ensure a secure, maintainable, and extensible platform.

ACKNOWLEDGMENTS

We would like to acknowledge and thank Michael Polinski for his continuous support, guidance, and collaboration throughout this work. His expertise, patience, and dedication were central to the progress and completion of this project. We are also grateful to Peter Dinda for his valuable insights and contributions.

REFERENCES

[1] John Lange, Peter Dinda, Robert Dick, Friedrich Doku, Elena Fabian, Nick Gordon, Peizhi Liu, Michael Polinski, Madhav Suresh, Carson Surmeier, and Nick Wanninger. 2023. *A Case for a User-centered Distributed Privacy Backplane for the Internet of Things.* Technical Report NU-CS-2023-09. Computer Science Department, Northwestern University. https://www.mccormick.northwestern.edu/computer-science/documents/nu-cs-2023-09.pdf Accessed: 2025-06-11.

[2] Protocol Labs. 2023. libp2p: Modular Network Stack. https://libp2p.io/. Accessed: 2025-06-11.