PrivacyRadar

Software Design Document
SDD 1.1

Authors:
David Osborne
Luke Therieau
Madhur Deep Jain
Yashna Praveen
Sujal Charak

Table of Contents

Introduction

1.1 Purpose

This document describes the detailed design of PrivacyRadar and its application-based network traffic capture system. It serves as the guide for implementation, integration, and maintenance of the software. It defines the design architecture, modules, interfaces, and data structures necessary to capture network packets, decode them, and associate them with active processes running on the host machine.

1.2 Scope

The system captures all inbound and outbound packets using the Cap library, decodes headers (Ethernet, IPv4 & IPv6, TCP/UDP) and identifies which local processes generated or received the network traffic. The output links each flow (socket or 5-tuple) to a process ID and name, providing a per-application traffic summary. This document focuses on the software design including module decomposition, data management, dependencies, and interface design.

1.3 Definitions, Acronyms, and Abbreviations

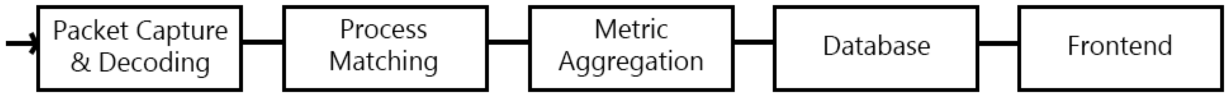
- *API:* Application Programming Interface
- *PID:* Process Identifier
- *UI:* User Interface
- *PCAP*: Packet Capture
- *Process*: An application instance actively running on the host machine that is associated with a port. It may be in a LISTENING state, and does not guarantee data is being transmitted or received.
- *Connection*: A process actively transmitting or receiving data across the network via an ESTABLISHED state. They are identified by having a unique 5-tuple communication stream (src/dst IP, src/dst ports, protocol).

References

- Software Requirements Specification (SRS) for PrivacyRadar
- Node.js v20 documentation
- Cap library documentation
- Ps-list library documentation
- OS process table APIs
- Wireshark Frame Decoding Reference

Decomposition Description

3.1 Module Decomposition



The system consists of three primary modules which operate in a pipeline. The Packet Capture module collects and decodes all network traffic from selected interfaces. The Process Matching module links each decoded packet with its originating or receiving application process using a series of lookup tables and caches via socket and PID information. Lastly, the Metrics Aggregation module utilizes enriched packet data to calculate structured statistics for persistent storage and visualization. Together these modules form the core data path from raw network capture to user-facing analytics.

The modules communicate through in-memory event streams. Each module operates independently and internal queues are used and managed to avoid packet loss when other modules are busy. This design allows the system to scale easily as each component can be optimized or extended independently without disrupting the packet processing pipeline.

3.1.1 Packet Capture Module

This module initializes network interfaces and captures raw packets using the Cap library. Its primary inputs are interface handles with capture parameters; essentially all raw network traffic associated with the interface. The outputs are decoded packet objects flushed to consumers. The core responsibilities of this module are to initialize the capture session, listen for traffic asynchronously, decode link-layer, network, and transport headers, and return structured packet data for further enrichment and analysis.

3.1.2 Process Matching Module

The process matching module links decoded packets with the process that owns the associated socket. The inputs are decoded packet headers (source/destination IP, port, and protocol) which it then uses to generate a lookup key for socket-to-process mapping. Outputs are enriched packet metadata containing process identifiers, names, and the full decoded ethernet frame. The core responsibilities of this module are to query active system socket tables, maintain an in-memory cache of socket-to-PID associations, and to update mappings when connections open or close.

3.1.3 Metrics Aggregation Module

The metrics aggregation module aggregates statistics from enriched packets to produce higher-level metrics. The inputs are fully decoded packets linked to processes, and the outputs are summarized metrics for database storage and frontend visualization. The main responsibilities include maintaining per-process and global counters (bytes sent/received, protocol usage, etc), computing time-windowed metrics for display, and forwarding a variety of structured metrics objects to the database or to the user interface.

3.2 Concurrent Process Description

Although PrivacyRadar runs on Node.js, which uses a single-threaded event loop, the system achieves concurrency through asynchronous, non-blocking I/O. The capture & decoding, process matching, and registry update operations execute as separate asynchronous tasks coordinated by the event loop. This ensures that continuous packet capture does not block decoding or database updates, maintaining smooth real-time performance.

3.2.1 Capture Loop

The capture loop is implemented as an event-driven listener using the Cap library. Each time a packet arrives, the *on('packet')* callback executes asynchronously. The core responsibilities of the capture loop are to read raw frames from the network interface buffer and decode headers using *PacketDecoder*, which incorporates the built-in decoding functionality of Cap. Packet decoding happens synchronously within the callback, but results are pushed to an in-memory queue. The queue acts as a producer output, and mutexes are not necessary since Node handles callbacks serially. Once packets have been decoded, they are enqueued as unenriched *PacketMetadata* objects for process matching.

3.2.2 Matching and Database Writes

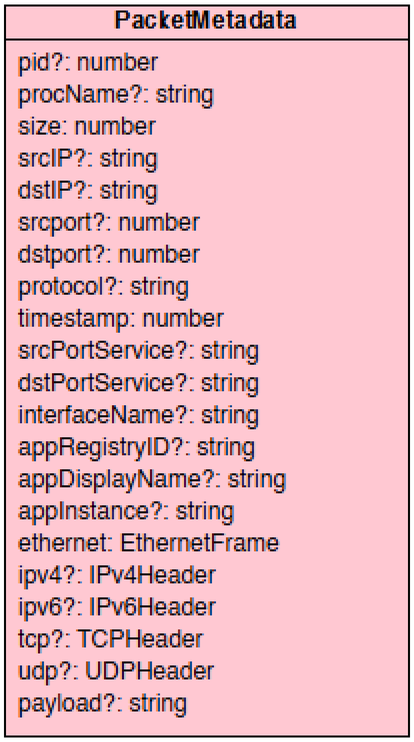
Process matching and metric aggregation occur asynchronously in scheduled tasks that consume packets from the queue. *ProcessTracker, ConnectionTracker,* and *ProcConManager* are used together to match packets to processes. Enriched *PacketMetadata* that includes the PID, PPID if applicable, and the process' name are then sent to the *ApplicationRegistryManager* which updates the global metrics for the network interface and to the specific process. These registries periodically send updates to the database to be reflected on the front end. For periodic batch operations, *setInterval()* is utilized. Database writes are I/O bound and are handled with *async/await* to prevent blocking packet capture. This producer-consumer model ensures packet capture always has priority in the event loop.

3.2.3 Synchronization Model

Node’s event loop guarantees that each callback executes to completion before the next event takes place. Shared data structures like the packet queue or registries are accessed sequentially without the need for thread-level synchronization. When necessary, atomic assignments or shallow object copies are used to prevent race conditions between capture and matching tasks.

3.3 Data Decomposition

3.3.1 PacketMetadata Interface



The diagram illustrates the standardized data representation used by all components that handle network packets. PacketMetadata is of type ‘Interface’ and is treated as a data structure contract rather than a complete class. It defines the structure for all captured and decoded packets in the system. Every class that handles network packets including *TrafficCapture*, *PacketDecoder*, and *ProcConManager* operate on objects conforming to this interface. The majority of the attributes contain the optional property modifier to account for the various types of network traffic that may be sent and received. For example, one process may be communicating over IPv4 using TCP, and another may be using UDP over IPv6; the headers will contain differing fields that all need to be accounted for. This design also allows for capture and recording of network/system control packets, which may not contain any IP addresses.

3.3.2 Process Mapping Tables

The system uses 4 mapping tables to aid in packet-to-process matching. Each table is designed to handle different aspects of network traffic and process tracking. Loopback interfaces can be ignored, since local-only traffic does not require external process association.

- *procCache* (ProcessTracker)
    - Stores all active processes and their associated metadata (PID, PPID, name, cpu usage, memory usage, and ports).
    - Populated using the ps-list library on the host OS.
    - Serves as the primary lookup table for associating a PID with a name.
- *connectionsList* (ConnectionTracker)
    - Maintains a list of all fully established TCP and UDP flows observed on the host.
    - Stored data consists of full 5-tuple entries (source IP, destination IP, source port, destination port, protocol) as well as the name of the associated process.
    - Serves as the primary reference for associating packets with network flows.
- *udpCache* (ConnectionTracker)
    - Temporarily stores ephemeral socket states for UDP traffic since UDP is connectionless.
    - Stored data consists of source port and, if available, source IP address, as well as the PID and timestamp of the last seen packet.
    - Frequently pruned to remove stale entries.
    - Enables packet matching for short-lived UDP endpoints that are difficult to track.
- tcpCache (ConnectionTracker)
    - Handles TCP flows that have not yet fully established or are in an intermediate state.
    - Stored data consists of source and destination IPs and ports, PID, and connection state.

3.3.3 Registry Structures

The registry structures store aggregated network activity data at both the interface and process levels. These structures are continuously updated by their respective registry module based on enriched packet metadata.

- *GlobalRegistry* tracks overall interface-level statistics such as total packets, bytes, remote destinations, and protocol breakdown.
- *ProcessRegistry* stores per-process statistics linked by PID and is used for frontend visualization to display per-application network activity, if the user chooses to filter by a specific process. *ProcessRegistry* objects are stored in
- *ApplicationRegistry* tracks processes linked by either the same application name or by a parent-child PID relationship.

Dependency Description

4.1 Intermodule Dependencies

PrivacyRadar's design uses a one-way data flow to minimize coupling and circular dependencies. All modules communicate through shared interfaces and queues; no direct global variable access occurs.

- *TrafficCapture* -> *Decoder*
  - Captured raw packet buffers are decoded immediately to extract headers and payloads
- *ProcessTracker* & *ConnectionTracker* -> *ProcConManager*
  - A table of active process IDs + name (ps-list) is cross-referenced with a table of process IDs with active network connections and stored separately. This table is used to match network traffic to processes.
- *NetworkAnalyzer* -> *ApplicationRegistry* & *GlobalRegistry*
  - Enriched network traffic (PID, name, 5-tuple, and data metrics) is stored in its respective registry.

4.2 Interprocess Dependencies

The main coordination mechanism is a producer-consumer queue: Captured packets are enqueued by the capture loop and consumed by the decoding, matching, and aggregation processes.

- The Traffic Capture process depends on the Node event loop for non-blocking I/O continuous packet collection.
- The Packet Matching process depends on the packet queue and consumes decoded packets produced by the capture loop in order.
- The Registry Update process depends on the output of the matching process to aggregate and persist metrics.

4.3 Data Dependencies

All shared data structures such as the packet queue, mapping tables, and registries are accessed sequentially under Node’s single-threaded event model. This eliminates the need for traditional synchronization techniques like mutexes. To prevent stale data and maintain consistency, each structure is updated atomically within its event cycle, utilizing shallow copies when necessary.

Interface Description

5.1 Module Interface

5.1.1 Packet Capture Module

Function: initialize the capture session, listen for packets, and decode raw buffers

Public API:
- *startCapture*(interfaceName: string): void
- *stopCapture*( ): void
- *onPacket*(callback: (packet: Buffer) => void): void
- flushQueue( ): void

Input: A Network Interface handle

Output: Raw packet buffers passed to the Decoder

Expected Behavior: Non-blocking, continuous capture of all inbound/outbound traffic on selected interface(s)

5.1.2 Process Matching Module

Function: Associate packets with system processes using socket and PID mappings

Public API:

- *matchPacket*(packet: PacketMetadata): Promise<PacketMetadata>
- refreshProcessList( ): Promise<void>

Input: Decoded packet metadata

Output: Enriched packet metadata including PID, PPID, and process name linked to the packet.

Expected Behavior: Uses the main lookup table first for a full 5-tuple connection, and then checks the caches (UDP or TCP) for partial matching. If both fail, the packet is returned with an “UNKNOWN” name tag.

5.2 Process Interface

This section describes the logical processes that execute concurrently within PrivacyRadar and how they interact. Although Node.js runs on a single thread, concurrency is achieved through asynchronous operations coordinated by the event loop. Each “process” in this context represents an asynchronous task responsible for part of the system’s workflow.

5.2.1 Capture Process

The Capture Process is an event-driven producer that interfaces directly with the Cap library to receive raw packet buffers from the network interface. It is responsible for decoding link-layer and network-layer headers (Ethernet, IPv4/IPv6, TCP/UDP) and constructing partial PacketMetadata objects. Each decoded packet is immediately enqueued for later enrichment.

Responsibilities:

- Initialize network capture on the selected interface.
- Decode raw frames into structured metadata.
- Maintain continuous capture with minimal packet loss.
- Push decoded packets into a shared, in-memory queue.

Communication:

- Incoming: None, the process operates continuously until manually stopped.
- Outgoing: Publishes *PacketMetadata* objects to the shared queue.

Concurrency Model:

- The Cap event listener executes asynchronously under Node's event loop. Since each callback completes before the next begins, no explicit mutexes are required. The queue serves as the handoff mechanism between capture and matching tasks.

5.2.2 Matching Process

The Matching Process consumes decoded packets from the shared queue, attempts to associate them with active processes using the ProcessTracker and ConnectionTracker, and enriches each packet with process information such as PID and name. It operates asynchronously and continuously, keeping up with the capture rate.

Responsibilities:

- Consume *PacketMetadata* objects from the queue.
- Query mapping tables (full connection table, then caches if necessary).
- Enrich packets with associated PID and process name.
- Forward enriched packets to the Metrics Aggregator, then to the database directly.

Communication:

- Incoming: Reads *PacketMetadata* objects from the shared queue
- Outgoing: Sends enriched metadata to the ApplicationRegistryManager

Concurrency Model:

- Runs as an asynchronous task scheduled with *setInterval( )* to process batches of packets. The event loop ensures ordered, non-blocking execution. No mutexes are needed, but shallow copies are used when updating shared structures to prevent race conditions.

5.2.3 Registry Update Process

The Registry Update Process collects enriched packets and aggregates metrics per process and per interface. It operates periodically (e.g., every few seconds) to compute byte counts, packet totals and percentages, and protocol distributions before committing results to the database.

Responsibilities:

- Consume enriched *PacketMetadata* from the Matching Process.
- Update global, application, and per-process registries.
- Write aggregated data to persistent storage.

Communication:

- Incoming: Receives enriched metadata from the Matching Process.
- Outgoing: Writes aggregated statistics to the database and/or to the front end directly.

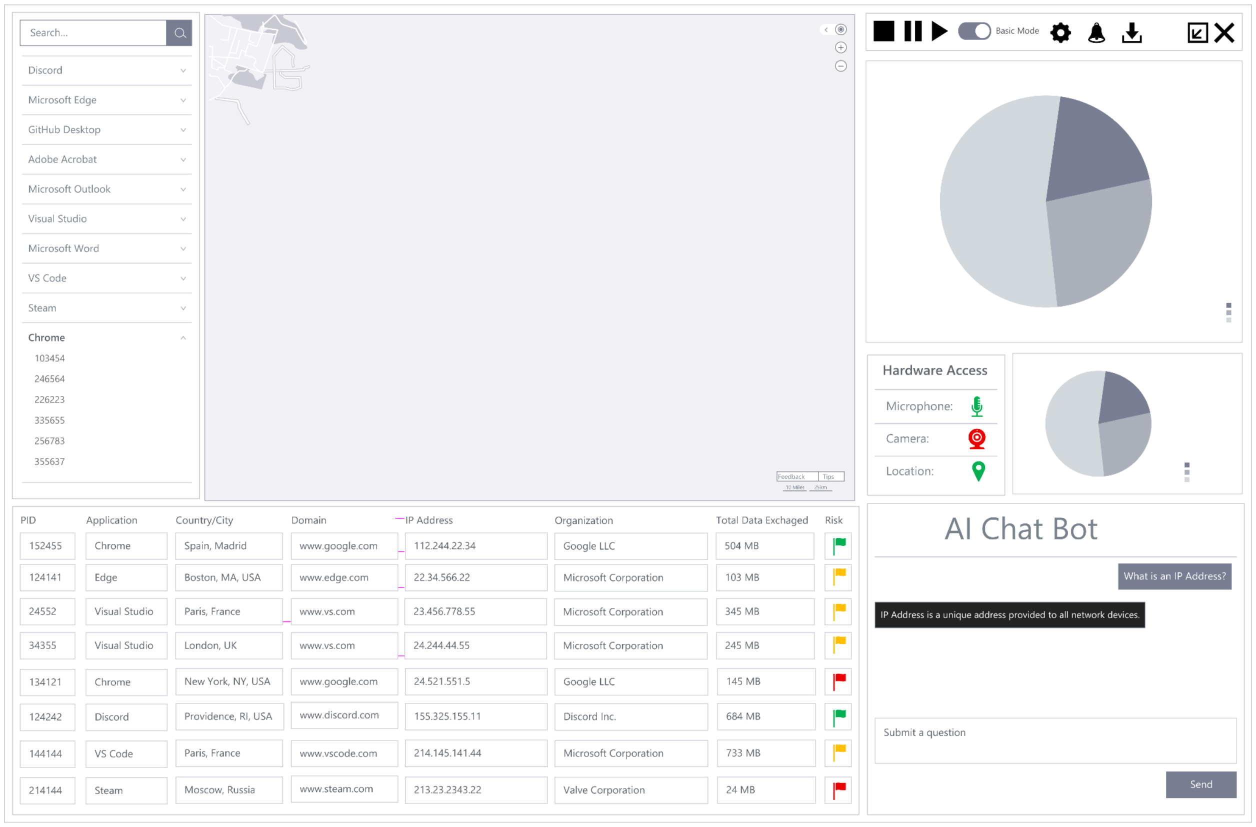
Concurrency Model:

- Operates asynchronously at a fixed interval using *setInterval( )*. Database writes are performed with async/await to avoid blocking the event loop.

5.3 User Interface

5.2.1 Wireframes

Basic Mode:

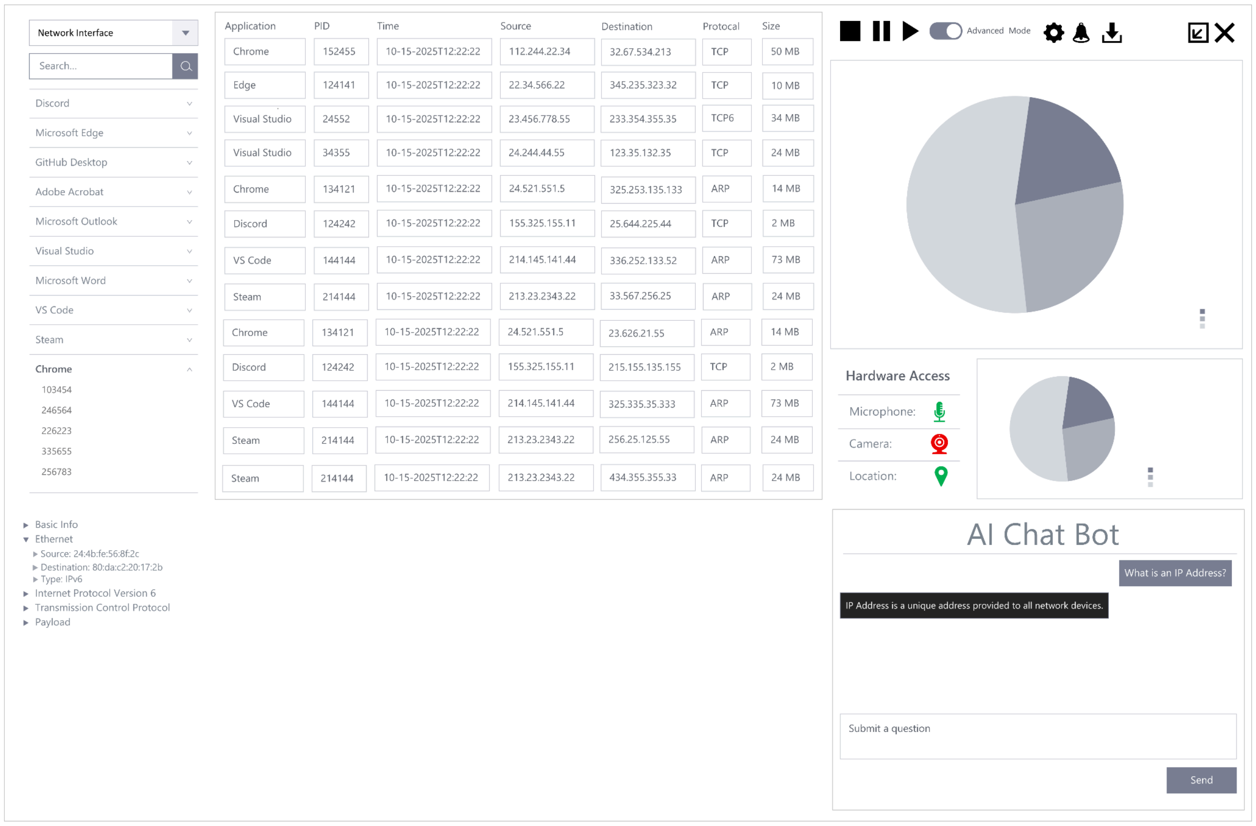


What is an IP Address?

IP Address is a unique address provided to all network devices.

PID	Application	Country/City	Domain	IP Address	Organization	Total Data Exchaged	Risk
152455	Chrome	Spain, Madrid	www.google.com	112.244.22.34	Google LLC	504 MB	🟩
124141	Edge	Boston, MA, USA	www.edge.com	22.34.566.22	Microsoft Corporation	103 MB	🟨
24552	Visual Studio	Paris, France	www.vs.com	23.456.778.55	Microsoft Corporation	345 MB	🟨
34355	Visual Studio	London, UK	www.vs.com	24.244.44.55	Microsoft Corporation	245 MB	🟨
134121	Chrome	New York, NY, USA	www.google.com	24.521.551.5	Google LLC	145 MB	🟥
124242	Discord	Providence, RI, USA	www.discord.com	155.325.155.11	Discord Inc.	684 MB	🟩
144144	VS Code	Paris, France	www.vscode.com	214.145.141.44	Microsoft Corporation	733 MB	🟨
214144	Steam	Moscow, Russia	www.steam.com	213.23.2343.22	Valve Corporation	24 MB	🟥

Advanced Mode:



Basic Info
Ethernet
Source: 24:4b:fe:56:8f:2c
Destination: 80:da:c2:20:17:2b
Type: IPv6
Internet Protocol Version 6
Transmission Control Protocol
Payload

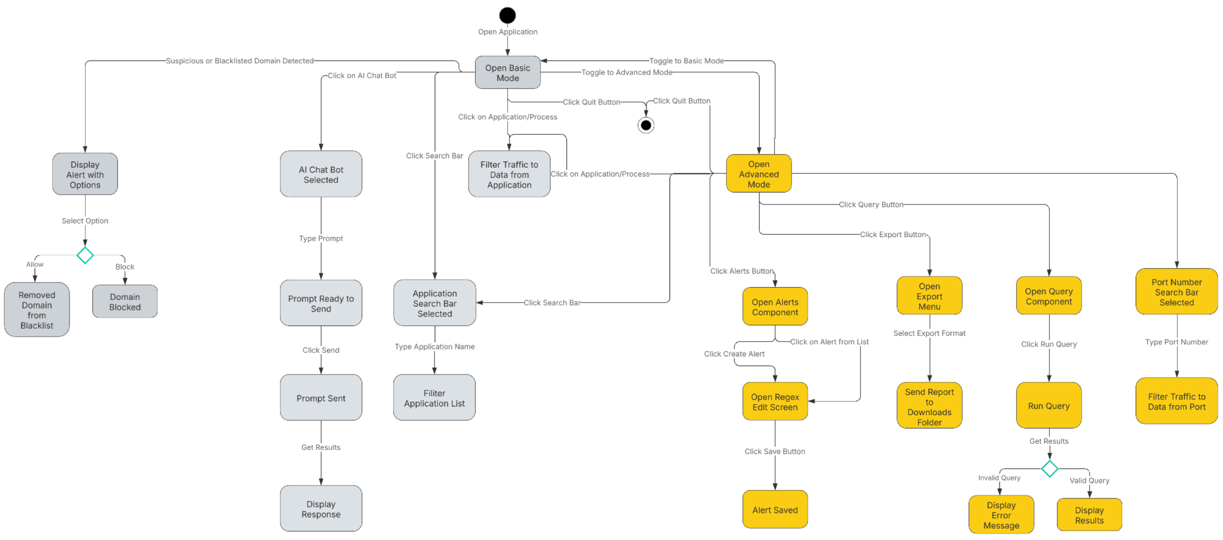
Application	PID	Time	Source	Destination	Protocal	Size
Chrome	152455	10-15-2025T12:22:22	112.244.22.34	32.67.534.213	TCP	50 MB
Edge	124141	10-15-2025T12:22:22	22.34.566.22	345.235.323.32	TCP	10 MB
Visual Studio	24552	10-15-2025T12:22:22	23.456.778.55	233.354.355.35	TCP6	34 MB
Visual Studio	34355	10-15-2025T12:22:22	24.244.44.55	123.35.132.35	TCP	24 MB
Chrome	134121	10-15-2025T12:22:22	24.521.551.5	325.253.135.133	ARP	14 MB
Discord	124242	10-15-2025T12:22:22	155.325.155.11	25.644.225.44	TCP	2 MB
VS Code	144144	10-15-2025T12:22:22	214.145.141.44	336.252.133.52	ARP	73 MB
Steam	214144	10-15-2025T12:22:22	213.23.2343.22	33.567.256.25	ARP	24 MB
Chrome	134121	10-15-2025T12:22:22	24.521.551.5	23.626.21.55	ARP	14 MB
Discord	124242	10-15-2025T12:22:22	155.325.155.11	215.155.135.155	TCP	2 MB
VS Code	144144	10-15-2025T12:22:22	214.145.141.44	325.335.35.333	ARP	73 MB
Steam	214144	10-15-2025T12:22:22	213.23.2343.22	256.25.125.55	ARP	24 MB
Steam	214144	10-15-2025T12:22:22	213.23.2343.22	434.355.355.33	ARP	24 MB

What is an IP Address?

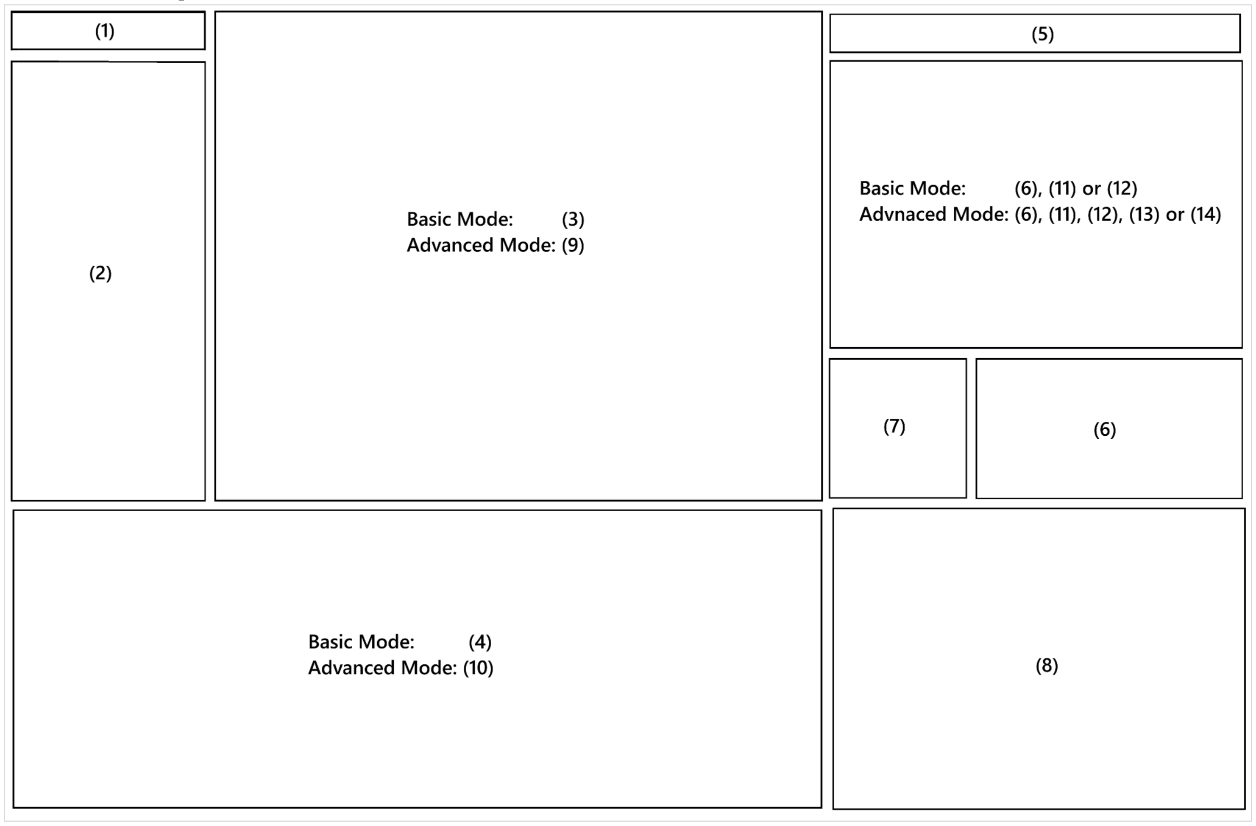
IP Address is a unique address provided to all network devices.

Submit a question

5.2.2 State Diagram



5.2.3 Components



Search Component (1):

Functionalities:

- Ability to search by application name.
- Ability to search by process id.
- Ability to search by port number.

User Stories:

- GU-NM-002: As a general user, I want to click on any application to see detailed connection information including websites visited and data transferred, presented in non-technical language.
- AU-TA-001: As an advanced user, I want to filter network connections by specific TCP/UDP ports to identify service-specific traffic patterns.

Application and Process Component (2):

Functionalities:

- List of applications and their respective processes.
- Ability to click on a specific application and process to filter dashboards and their underlying traffic data.

Global Choropleth Map Component (3):

Functionalities:

- Visually represent the destination countries of all network traffic.
- Ability to filter dashboards and their underlying traffic data by country.

User Stories:

- GU-NM-003: As a general user, I want to see a world map highlighting countries where my data is being sent, with privacy risk indicators for each location.

Detailed Information Component (4):

Functionalities:

- Ability to view basic network information about applications and processes.

User Stories:

- GU-NM-002: As a general user, I want to click on any application to see detailed connection information including websites visited and data transferred, presented in non-technical language.
- GU-PP-002: As a general user, I want a privacy score for each application based on its behavior, helping me understand which apps respect my privacy.

Navigation Component (5):

Functionalities:

- Start, stop & pause functionality for network monitoring.
- Toggle switch for navigation between advanced and basic mood.
- Buttons to open Settings, Alerts, Export, or Query Components.

Summary Dashboard Component (6):

Functionalities:

- Line, bar or pie charts that summarize real time and historical network traffic.

User Stories:

- GU-NM-001: As a general user, I want to see a summary dashboard showing total network usage since PrivacyRadar started, broken down by application, so that I can quickly identify which apps consume the most bandwidth.
- GU-PP-003: As a general user, I want to see a timeline view of when applications accessed privacy-sensitive features throughout the day.

Hardware Access Component (7):

Functionalities:
- Real time indicators for the status of a system's camera, microphone and location.

AI Chatbot Component (8):

Functionalities:
- Ability to send questions and receive answers.

User Stories:
- GU-AI-001: As a general user, I want to ask the AI assistant questions like "Which apps sent data to China today?" and receive clear, actionable answers.
- GU-AI-003: As a general user, I want weekly AI-generated privacy reports summarizing important findings and recommendations.

Advanced Information Component (9):

Functionalities:
- Ability to see advanced information at the packet level.

User Stories:
- AU-TA-002: As an advanced user, I want to see raw packet count and byte-level statistics for each connection.
- AU-TA-003: As an advanced user, I want to correlate process IDs with network connections for forensic analysis.

Advanced Detailed Information Component (10):

Functionalities:
- Ability to see more detailed information about a single selected packet.

Settings Component (11):

Functionalities:
- Ability to configure data retention.
- Ability to toggle between dark and light mode.
- Ability to change the default mode (Basic or Advanced).

User Stories:
- AU-DM-003: As an advanced user, I want to configure data retention policies to automatically purge old records based on my requirements.

Notifications and Alerts Component (12):

Functionalities:

- Ability to view a list of notifications.
- Ability to perform actions on specific messages (Allow/Block/Learn).
- Ability to customize alerting rules via regex.

User Stories:

- GU-PP-001: As a general user, I want to receive immediate pop-up notifications when any application accesses my camera or microphone, with options to investigate further or take action.
- GU-PP-004: As a general user, I want to receive an alert when an application connects to a suspicious or blacklisted domain, with a short explanation and options to Allow/Block/Learn More, so that I can protect my data from unsafe connections and stay in control of my privacy.
- GU-AI-002: As a general user, I want the AI to proactively suggest privacy improvements based on detected patterns in my application usage.
- AU-TA-004: As an advanced user, I want to create custom alert rules using regex patterns for domain names and IP ranges.

Export Component (13):

Functionalities:

- Ability to export raw network data into various file formats (xlsx, csv, pdf).
- Ability to export previous queries.

User Stories:

- AU-DM-001: As an advanced user, I want to export monitoring data.

Query Component (14):

Functionalities:

- Ability to use SQL like queries for analysis.

User Stories:

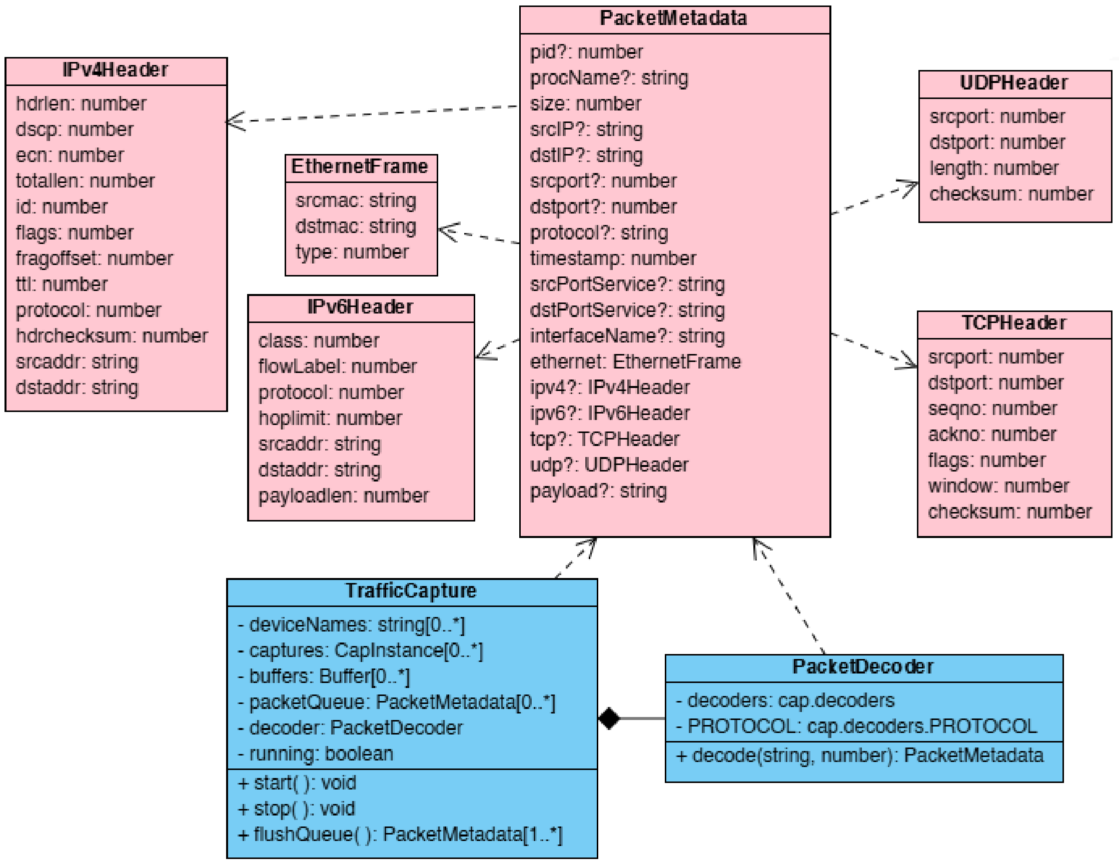
- AU-DM-002: As an advanced user, I want to create custom SQL queries against the monitoring database for specialized analysis.

Detailed Design

6.1 Module Detailed Design

6.1.1 Packet Capture Module Design

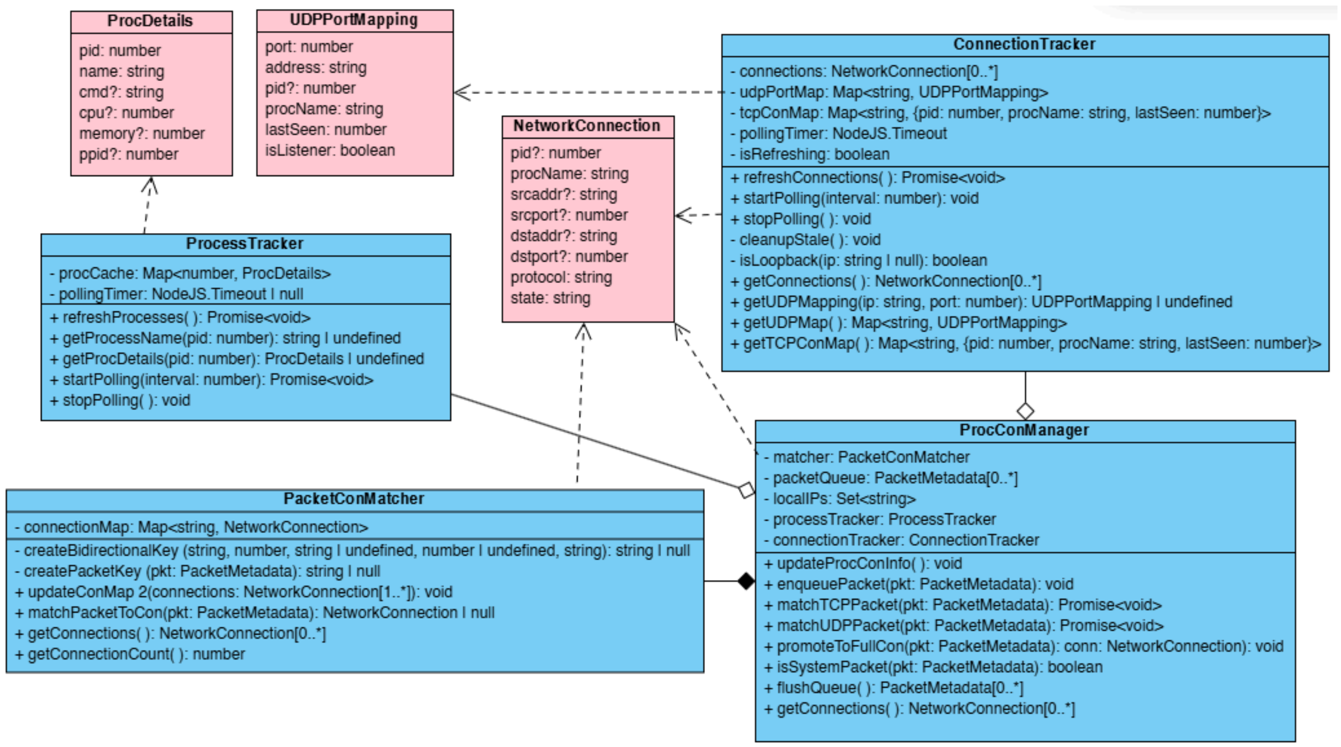
Note: Interfaces (type contracts) are colored in pink, with core classes colored in blue.



Flow:

1. Initialize Cap and open network interface
2. Attach event listener: *cap.on*(‘packet’, callback)
3. Decode Ethernet, IP, and transport headers using *PacketDecoder*
4. Store decoded traffic in an in-memory queue for further processing
    - *flushQueue( )* is the public method to retrieve decoded the packet buffer from Traffic Capture

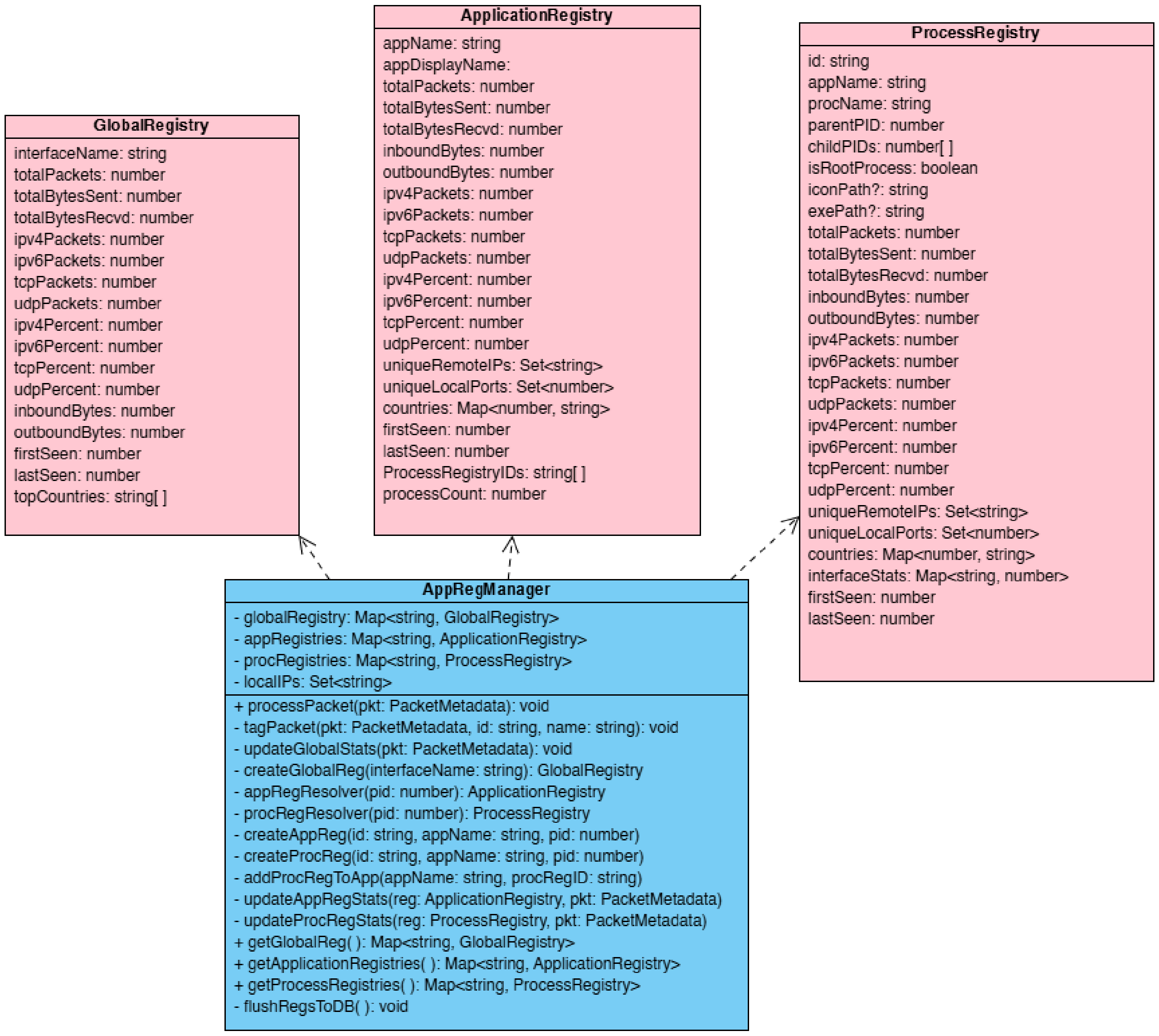
6.1.2 Process Matching Module Design


ProcDetails
pid: number
name: string
cmd?: string
cpu?: number
memory?: number
ppid?: number
UDPPortMapping
port: number
address: string
pid?: number
procName: string
lastSeen: number
isListener: boolean
ConnectionTracker
- connections: NetworkConnection[0..*]
- udpPortMap: Map<string, UDPPortMapping>
- tcpConMap: Map<string, {pid: number, procName: string, lastSeen: number}>
- pollingTimer: NodeJS.Timeout
- isRefreshing: boolean
+ refreshConnections( ): Promise<void>
+ startPolling(interval: number): void
+ stopPolling( ): void
- cleanupStale( ): void
- isLoopback(ip: string I null): boolean
+ getConnections( ): NetworkConnection[0..*]
+ getUDPMapping(ip: string, port: number): UDPPortMapping I undefined
+ getUDPMap( ): Map<string, UDPPortMapping>
+ getTCPConMap( ): Map<string, {pid: number, procName: string, lastSeen: number}>
NetworkConnection
pid?: number
procName: string
srcaddr?: string
srcport?: number
dstaddr?: string
dstport?: number
protocol: string
state: string
ProcessTracker
- procCache: Map<number, ProcDetails>
- pollingTimer: NodeJS.Timeout I null
+ refreshProcesses( ): Promise<void>
+ getProcessName(pid: number): string I undefined
+ getProcDetails(pid: number): ProcDetails I undefined
+ startPolling(interval: number): Promise<void>
+ stopPolling( ): void
ProcConManager
- matcher: PacketConMatcher
- packetQueue: PacketMetadata[0..*]
- localIPs: Set<string>
- processTracker: ProcessTracker
- connectionTracker: ConnectionTracker
+ updateProcConInfo( ): void
+ enqueuePacket(pkt: PacketMetadata): void
+ matchTCPPacket(pkt: PacketMetadata): Promise<void>
+ matchUDPPacket(pkt: PacketMetadata): Promise<void>
+ promoteToFullCon(pkt: PacketMetadata): conn: NetworkConnection): void
+ isSystemPacket(pkt: PacketMetadata): boolean
+ flushQueue( ): PacketMetadata[0..*]
+ getConnections( ): NetworkConnection[0..*]
PacketConMatcher
- connectionMap: Map<string, NetworkConnection>
- createBidirectionalKey (string, number, string I undefined, number I undefined, string): string I null
- createPacketKey (pkt: PacketMetadata): string I null
+ updateConMap 2(connections: NetworkConnection[1..*]): void
+ matchPacketToCon(pkt: PacketMetadata): NetworkConnection I null
+ getConnections( ): NetworkConnection[0..*]
+ getConnectionCount( ): number

Flow:

1. Pop PacketMetadata from queue
2. Check connectionsList for existing 5-Tuple bidirectional key
    - If not found, check tcpConMap or udpPortMap (caches)
    - If found in caches, upgrade to connectionsList as a new connection
3. Enrich packet metadata with PID and process name
4. Forward to ApplicationRegistryManager for metric aggregation (shown below)

6.1.3 Metrics Aggregation Module Design

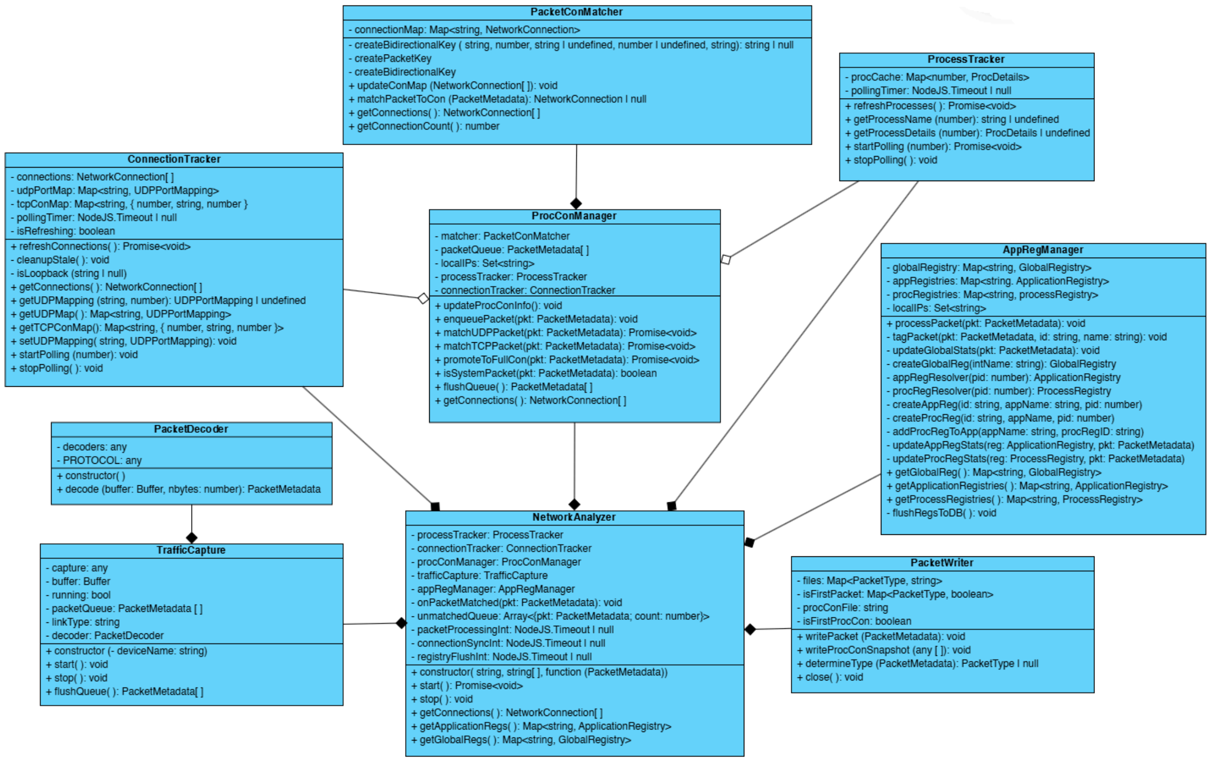


Flow:

1. Receive enriched packets from NetworkAnalyzer.
2. Update GlobalRegistry with each new packet for interface-wide stats and metrics.
3. Update the appropriate ApplicationRegistry for stats and metrics linked to the packet.
4. Update the specific ProcessRegistry for specific stats and metrics linked to the packet.
5. Periodically flush all three registries to the database.

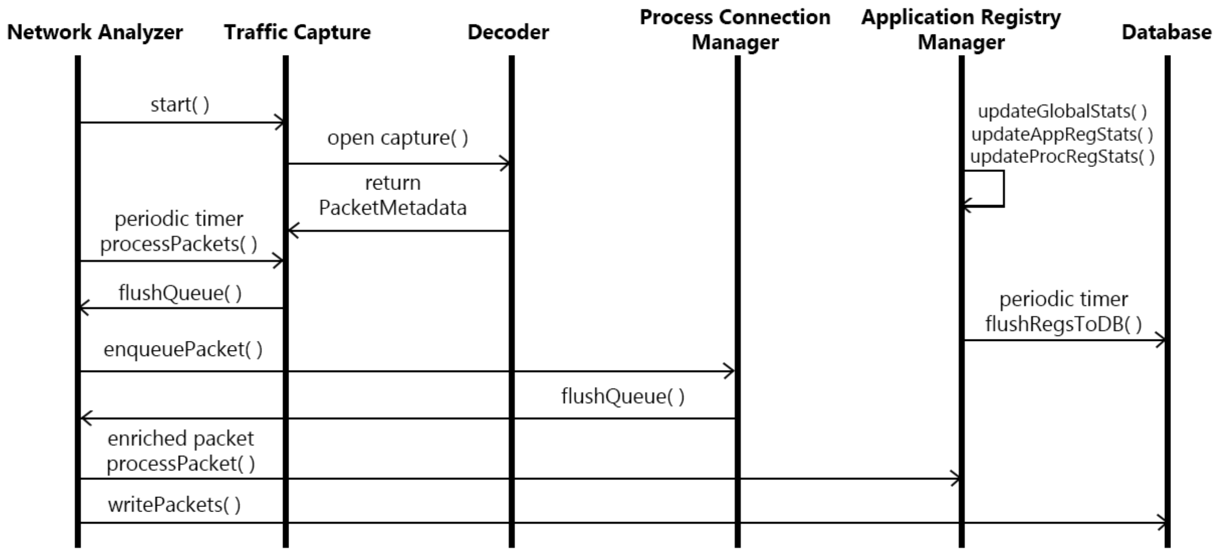
6.2 Data Detailed Design

6.2.1 Data Structures


PacketConMatcher
- connectionMap: Map<string, NetworkConnection>
- createBidirectionalKey (string, number, string I undefined, number I undefined, string): string I null
- createPacketKey
- createBidirectionalKey
+ updateConMap (NetworkConnection[ ]): void
+ matchPacketToCon (PacketMetadata): NetworkConnection I null
+ getConnections( ): NetworkConnection[ ]
+ getConnectionCount( ): number
ProcessTracker
- procCache: Map<number, ProcDetails>
- pollingTimer: NodeJS.Timeout I null
+ refreshProcesses( ): Promise<void>
+ getProcessName (number): string I undefined
+ getProcessDetails (number): ProcDetails I undefined
+ startPolling (number): Promise<void>
+ stopPolling( ): void
ConnectionTracker
- connections: NetworkConnection[ ]
- udpPortMap: Map<string, UDPPortMapping>
- tcpConMap: Map<string, { number, string, number }
- pollingTimer: NodeJS.Timeout I null
- isRefreshing: boolean
+ refreshConnections( ): Promise<void>
- cleanupStale( ): void
- isLoopback (string I null)
+ getConnections( ): NetworkConnection[ ]
+ getUDPMapping (string, number): UDPPortMapping I undefined
+ getUDPMap( ): Map<string, UDPPortMapping>
+ getTCPConMap(): Map<string, { number, string, number }>
+ setUDPMapping(string, UDPPortMapping): void
+ startPolling (number): void
+ stopPolling( ): void
ProcConManager
- matcher: PacketConMatcher
- packetQueue: PacketMetadata[ ]
- localIPs: Set<string>
- processTracker: ProcessTracker
- connectionTracker: ConnectionTracker
+ updateProcConInfo(): void
+ enqueuePacket(pkt: PacketMetadata): void
+ matchUDPPacket(pkt: PacketMetadata): Promise<void>
+ matchTCPPacket(pkt: PacketMetadata): Promise<void>
+ promoteToFullCon(pkt: PacketMetadata): Promise<void>
+ isSystemPacket(pkt: PacketMetadata): boolean
+ flushQueue( ): PacketMetadata[ ]
+ getConnections( ): NetworkConnection[ ]
AppRegManager
- globalRegistry: Map<string, GlobalRegistry>
- appRegistries: Map<string. ApplicationRegistry>
- procRegistries: Map<string, processRegistry>
- localIPs: Set<string>
+ processPacket(pkt: PacketMetadata): void
- tagPacket(pkt: PacketMetadata, id: string, name: string): void
- updateGlobalStats(pkt: PacketMetadata): void
- createGlobalReg(intName: string): GlobalRegistry
- appRegResolver(pid: number): ApplicationRegistry
- procRegResolver(pid: number): ProcessRegistry
- createAppReg(id: string, appName: string, pid: number)
- createProcReg(id: string, appName, pid: number)
- addProcRegToApp(appName: string, procRegID: string)
- updateAppRegStats(reg: ApplicationRegistry, pkt: PacketMetadata)
- updateProcRegStats(reg: ProcessRegistry, pkt: PacketMetadata)
+ getGlobalReg( ): Map<string, GlobalRegistry>
+ getApplicationRegistries( ): Map<string, ApplicationRegistry>
+ getProcessRegistries( ): Map<string, ProcessRegistry>
- flushRegsToDB( ): void
PacketDecoder
- decoders: any
- PROTOCOL: any
+ constructor( )
+ decode (buffer: Buffer, nbytes: number): PacketMetadata
TrafficCapture
- capture: any
- buffer: Buffer
- running: bool
- packetQueue: PacketMetadata [ ]
- linkType: string
- decoder: PacketDecoder
+ constructor (- deviceName: string)
+ start( ): void
+ stop( ): void
+ flushQueue( ): PacketMetadata[ ]
NetworkAnalyzer
- processTracker: ProcessTracker
- connectionTracker: ConnectionTracker
- procConManager: ProcConManager
- trafficCapture: TrafficCapture
- appRegManager: AppRegManager
- onPacketMatched(pkt: PacketMetadata): void
- unmatchedQueue: Array<{pkt: PacketMetadata; count: number}>
- packetProcessingInt: NodeJS.Timeout I null
- connectionSyncInt: NodeJS.Timeout I null
- registryFlushInt: NodeJS.Timeout I null
+ constructor(string, string[ ], function (PacketMetadata))
+ start( ): Promise<void>
+ stop( ): void
+ getConnections( ): NetworkConnection[ ]
+ getApplicationRegs( ): Map<string, ApplicationRegistry>
+ getGlobalRegs( ): Map<string, GlobalRegistry>
PacketWriter
- files: Map<PacketType, string>
- isFirstPacket: Map<PacketType, boolean>
- procConFile: string
- isFirstProcCon: boolean
+ writePacket (PacketMetadata): void
+ writeProcConSnapshot (any [ ]): void
+ determineType (PacketMetadata): PacketType I null
+ close( ): void

This diagram illustrates the primary modules (*TrafficCapture*, *ProcConManager*, *AppRegManager*, and *NetworkAnalyzer* classes) and their key attributes. *PacketMetadata* serves as the central data object, passed between modules and enriched at each stage of processing.

The *NetworkAnalyzer* class serves as the central coordinator for PrivacyRadar's packet processing pipeline. It does not introduce new functionality on its own but instead orchestrates the interactions between the core modules. It manages asynchronous timers for packet processing, connection synchronization, and registry updates. *NetworkAnalyzer* ultimately binds these modules together rather than being a separate module in the data path.



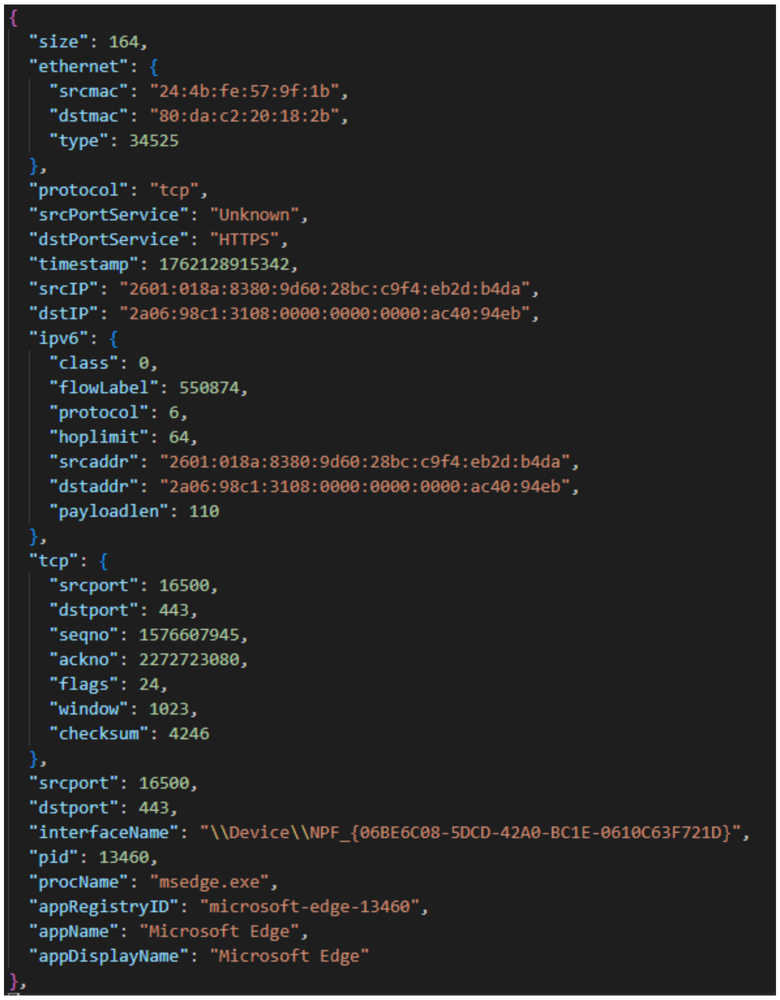
The sequence diagram shows the lifecycle of a PacketMetadata object through the system. *NetworkAnalyzer*.*start( )* initiates the system, opening the selected interfaces and beginning packet capture. Captured packets are decoded and stored in *TrafficCapture's* buffer, then periodically flushed to *NetworkAnalyzer* and enqueued to *ProcConManager*. Once linked to a process, enriched packets are returned to *NetworkAnalyzer*, which forwards them to *AppRegManager* to update global, application, and process statistics. These registries are periodically flushed to the database. The packets themselves are then stored in a separate database via the *writePackets( )* method, a public method for the *PacketWriter* class called by *NetworkAnalyzer*.

6.2.2 File Format

PrivacyRadar stores packet-level and aggregated metrics data in JSON format for storage in the database. JSON was chosen for lightweight structure, and ease of integration with both the Node.js backend and frontend visualization components. This format also provides consistent data fields across different modules while supporting optional or conditional fields, which are necessary to handle the wide range of header structures in network traffic. Two representative JSON structures are shown below:

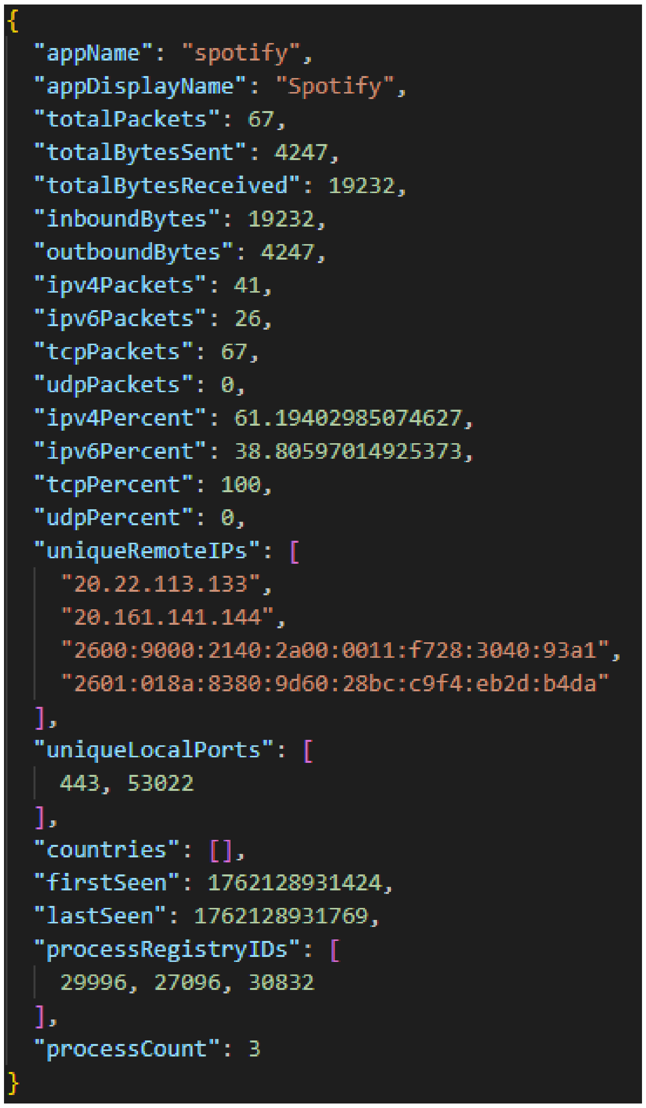
1. **PacketMetadata Object** - represents a single captured and enriched network packet. Each field corresponds to a property defined in the PacketMetadata interface including timestamp, source/destination IP addresses and ports,

associated PID, the process' name, and payload size. While full payloads are captured, only header information and metadata persist to reduce storage overhead. This structure is used for both temporary in-memory processing and persistence in the database.



```
{
  "size": 164,
  "ethernet": {
    "srcmac": "24:4b:fe:57:9f:1b",
    "dstmac": "80:da:c2:20:18:2b",
    "type": 34525
  },
  "protocol": "tcp",
  "srcPortService": "Unknown",
  "dstPortService": "HTTPS",
  "timestamp": 1762128915342,
  "srcIP": "2601:018a:8380:9d60:28bc:c9f4:eb2d:b4da",
  "dstIP": "2a06:98c1:3108:0000:0000:0000:ac40:94eb",
  "ipv6": {
    "class": 0,
    "flowLabel": 550874,
    "protocol": 6,
    "hoplimit": 64,
    "srcaddr": "2601:018a:8380:9d60:28bc:c9f4:eb2d:b4da",
    "dstaddr": "2a06:98c1:3108:0000:0000:0000:ac40:94eb",
    "payloadlen": 110
  },
  "tcp": {
    "srcport": 16500,
    "dstport": 443,
    "seqno": 1576607945,
    "ackno": 2272723080,
    "flags": 24,
    "window": 1023,
    "checksum": 4246
  },
  "srcport": 16500,
  "dstport": 443,
  "interfaceName": "\\Device\\NPF_{06BE6C08-5DCD-42A0-BC1E-0610C63F721D}",
  "pid": 13460,
  "procName": "msedge.exe",
  "appRegistryID": "microsoft-edge-13460",
  "appName": "Microsoft Edge",
  "appDisplayName": "Microsoft Edge"
},
```

2. **ApplicationRegistry Object** - represents per-application aggregated metrics. Fields include the application name for the associated processes, an array of process registry IDs for association, and a variety of cumulative metrics for all processes linked to a specific application. JSON field names are consistent across all Global, Application, and Process Registry objects which allows reliable parsing and aggregation.



```json
{
  "appName": "spotify",
  "appDisplayName": "Spotify",
  "totalPackets": 67,
  "totalBytesSent": 4247,
  "totalBytesReceived": 19232,
  "inboundBytes": 19232,
  "outboundBytes": 4247,
  "ipv4Packets": 41,
  "ipv6Packets": 26,
  "tcpPackets": 67,
  "udpPackets": 0,
  "ipv4Percent": 61.19402985074627,
  "ipv6Percent": 38.80597014925373,
  "tcpPercent": 100,
  "udpPercent": 0,
  "uniqueRemoteIPs": [
    "20.22.113.133",
    "20.161.141.144",
    "2600:9000:2140:2a00:0011:f728:3040:93a1",
    "2601:018a:8380:9d60:28bc:c9f4:eb2d:b4da"
  ],
  "uniqueLocalPorts": [
    443, 53022
  ],
  "countries": [],
  "firstSeen": 1762128931424,
  "lastSeen": 1762128931769,
  "processRegistryIDs": [
    29996, 27096, 30832
  ],
  "processCount": 3
}
```