Implementation of an Open World Recognition-capable Network Intrusion Detection System
Software Architecture Document

1. Introduction

This Software Architecture Document describes how the project "Implementation of an Open World Recognition-capable Network Intrusion Detection System" (IOWR-NIDS) is implemented in structure and code. This document will show how each function of the IOWR-NIDS system interacts with the rest.

The scope of this document is the interface, data collection, and packet tagging parts of the implementation. This will not include the inner functions of the neural network within the implementation as that is within the scope of the research, not the implementation.

This document will use the following terms and abbreviations, more information and terms are available in the Glossary document included in the project files:

Interface – A implementation of Dash to display various metrics and outputs of the model.

Model – An implementation of the artificial neural network with unknown detection seen here.

Client – The packet sniffer and the interface.

Server – The database, data collector, and model.

Network - The medium for which the client and server will communicate and sniff packets from.

2. Architectural Representation

This document provides several views on the system being developed. Because of the different viewpoints within this document some names of systems and subsystems may be inconsistent as they may refer to slightly different groups of components. This should not significantly limit understanding of this document.

The views of the system included in this document are:

- Use-case view: A diagram of the basic functionality required by the system and which subsystem initiates each function.
- Logical view 1: This is a package diagram that shows how things are contained in the project.
- Logical view 2: Provides a more in-depth view of the contents of each package with pseudo-class diagrams explaining the functionality of classes.

- Use case realization: A process flow diagram showing the heavyweight processes of the system and how the subsystems interact for each of those processes.
- Process view: Shows the lightweight processes of the system that are contained within the use case realization.
- Deployment view: Shows subsystem connection types and contents.
- Implementation view: Shows location of system components on computer instances.

3. Architectural Goals and Constraints

The server should be able to maintain multiple client connections at once, and multiple models along with it. Connections should be secured in some way to prevent data from being read by external actors. A Linux machine should be able to act as the server given a Python install and a PostgreSQL database that's been established with the provided schemas.

The client should be portable enough to spin up like a normal, native program.

The goals for the program include:

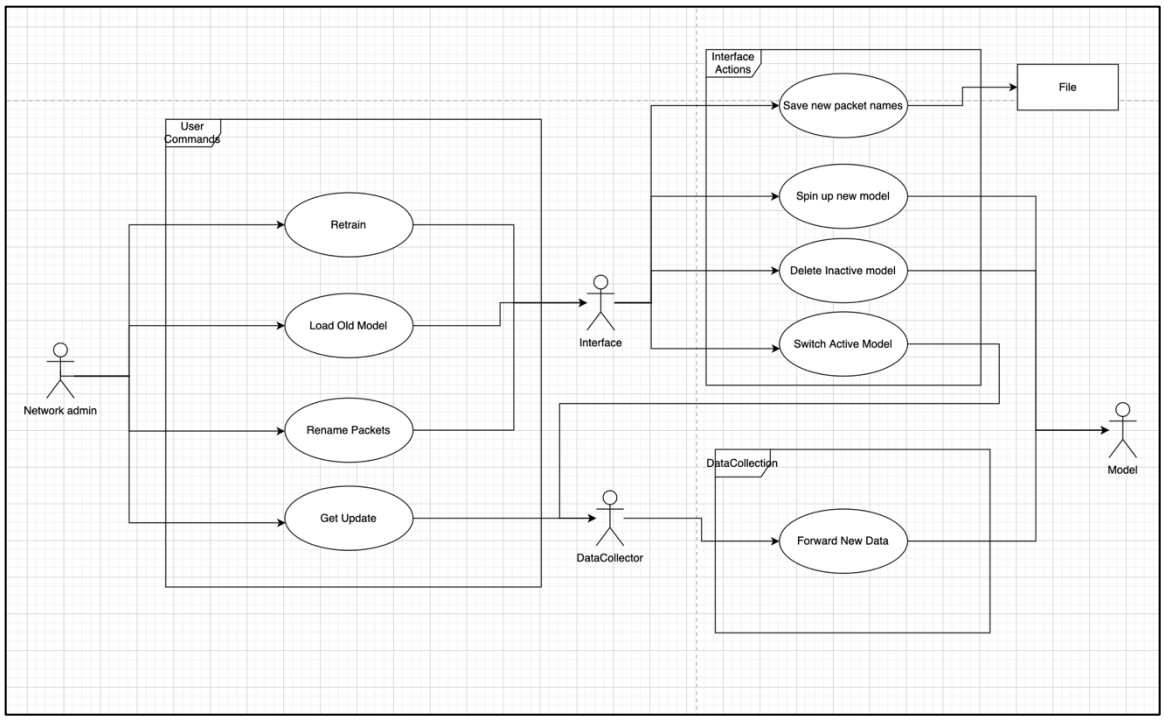
- Scanning packets from the network.
- Analyzing packets to classify them with the model.
- Displaying comprehensive metrics and data for the packets.
- Feeding training data, retraining, and swapping out the model.

The development process will be completed using GitHub's assortment of tools, including their Git hosting and Kanban-style Actions issue management. CI/CD is a foremost priority to ensure the overall stability of the main branch. UML will continue to be implemented with draw.io. Code quality will be additionally insured by Flake8, a linter for Python that dictates optimal/idiomatic formatting. All of these have varying levels of impact on the code architecture.

4. Use-Case View

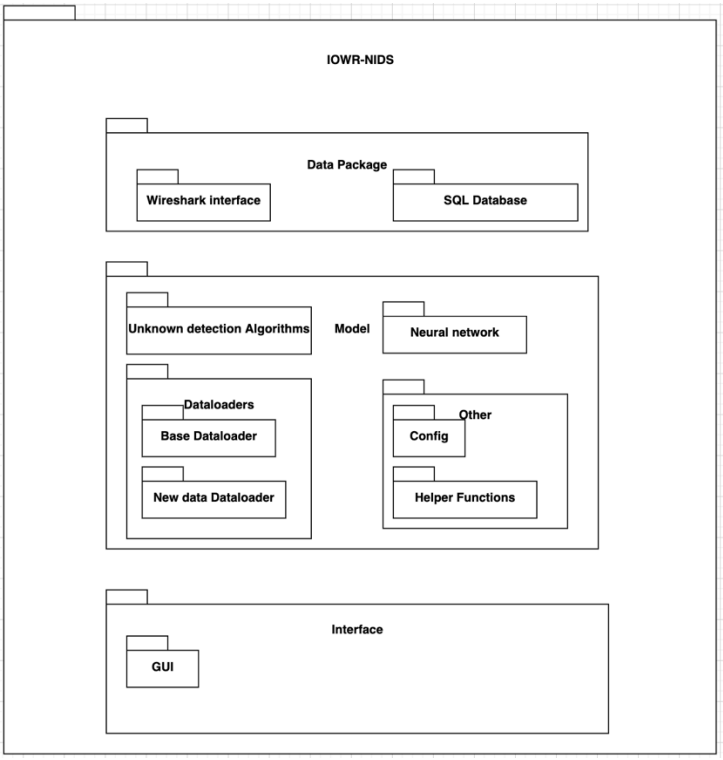
Main use cases:

1. Network Administrator wants to view the incoming packets.
2. Network Administrator wants to get an update for the packets.
3. Network Administrator wants to change out the model.

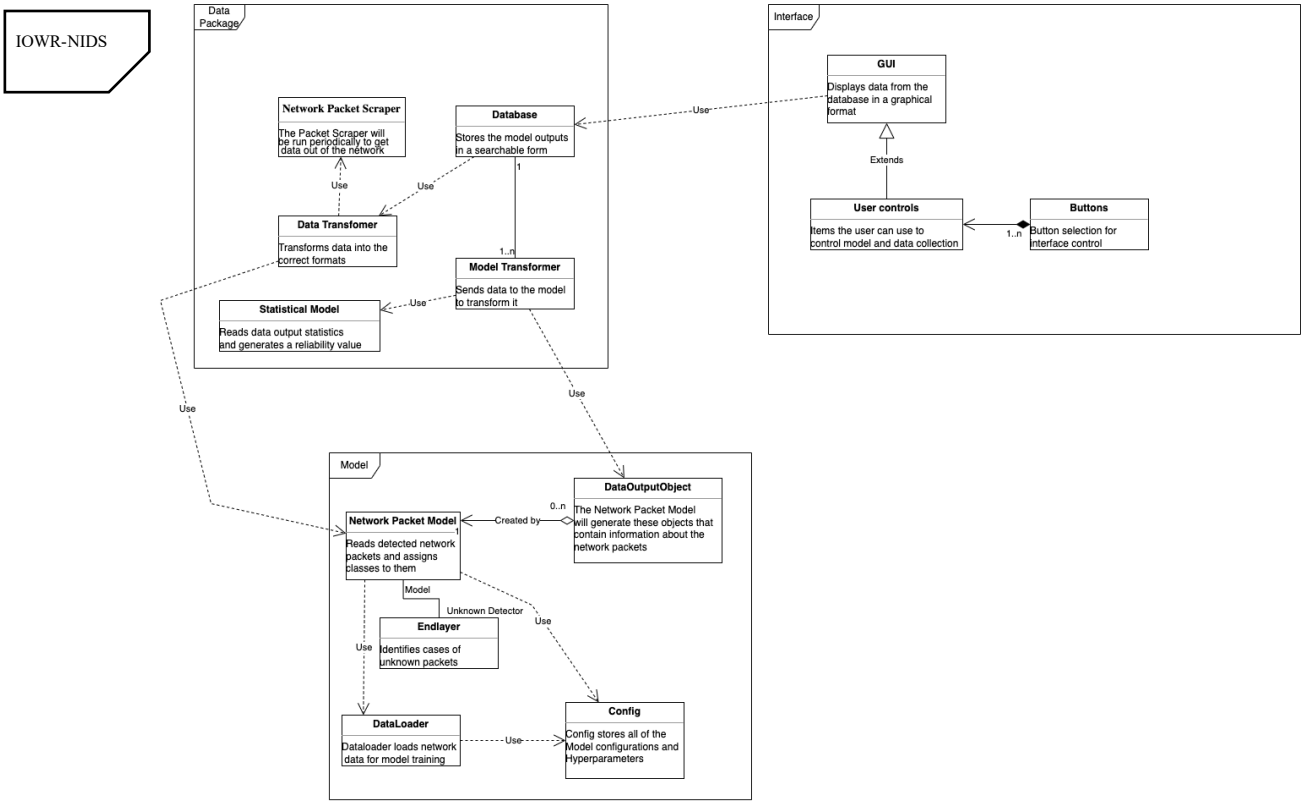


This is the use case view, it shows the network admin and the four base actions that they can take, Retrain, Load Old Model, Rename Packets, and Get Update.

5. Logical View



5.1 Overview



There are three main packages within the project. The Interface using Dash, the Data Package that collects and stores data, and the Model that classifies the data. Each of these are made of classes with descriptions seen here. However, there is no strict package hierarchy in the project and none of the packages have true sub packages.

5.2 Architecturally Significant Design Packages

GUI:

- The main interface a user will interact with.
- Displays packet data, model usage, classifications, integrity.
- User can select a different model or retrain a new model.

Network Packet Model:

- Classifies packets and provides confidence.
- Generates DataOutputObjects containing the output of the model.

DataOutputObject:

- Created by model to encapsulate all necessary data from the model output.

Network Packet Scraper:

- Sniffs network packets.
- Controls which network is scanned, and which packets are tracked.

Data Transform:

- Converts data between various formats needed internally.
- Batches data
- Handles database queries and inter-module communication.

Model Transform:

- Reads generated DataOutputObjects and saves the information in the database.

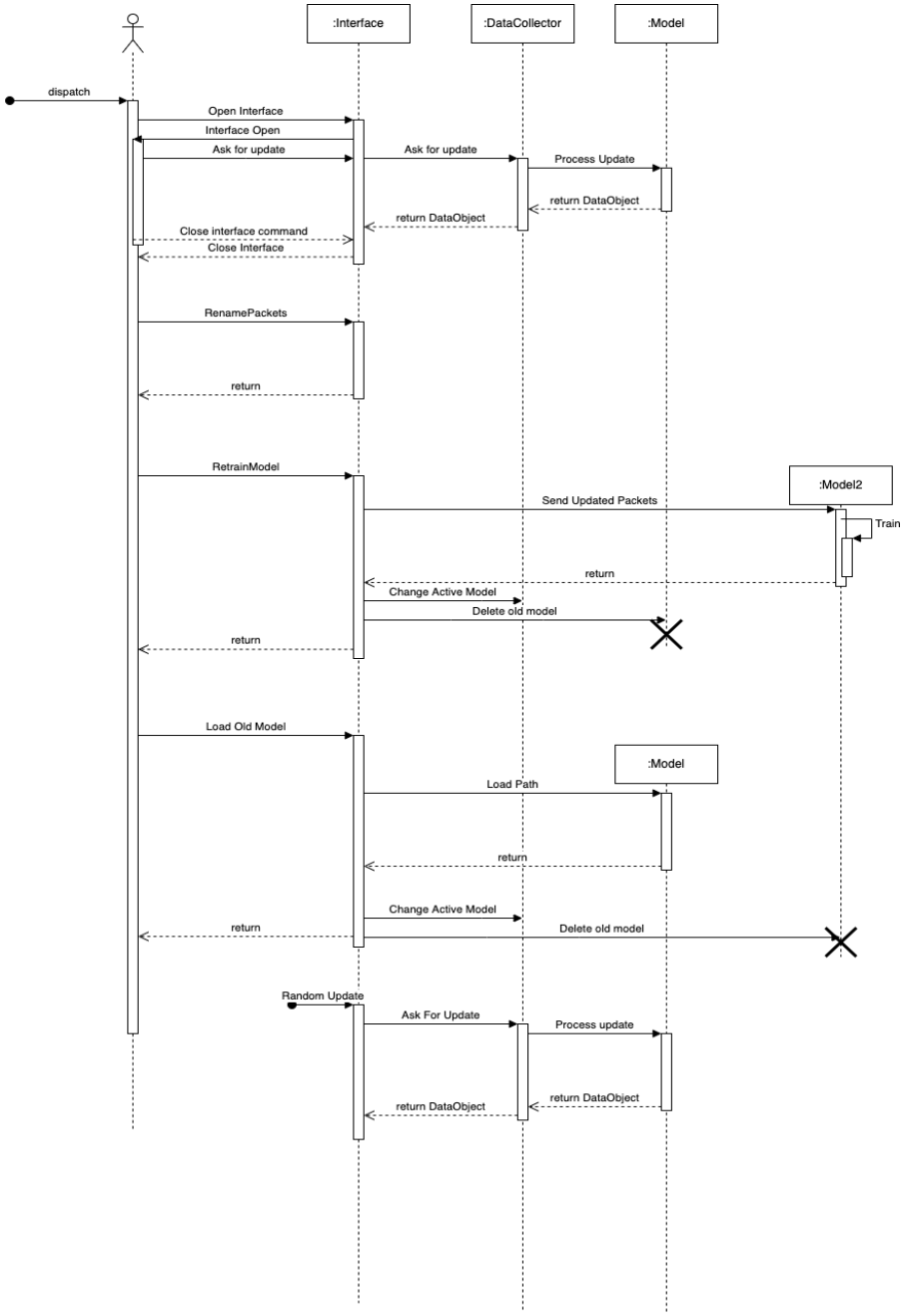
Database:

- Stores packets, classifications, and models
- Can be accessed by multiple users concurrently.

Statistics:

- Analyzes model to determine integrity.
- Details are currently unknown. Waiting on research to produce results and plan further.

5.3 Use-Case Realizations



These are the base use cases for the Network administrator and one-use case without the network administrator.

The first use case starting with the transition “Open interface” is the use case for the network administrator wanting to get an update. When the network administrator wants to get an update, they first open the interface which starts the interface being active and then sends the command to start the update. That update request gets propagated to the DataCollector, seen as Data Package in the package view, that collects the most recent data from the network (not pictured, out of scope) and then internally reformats that data and sends it to the model for

processing. Once the model has processed the data, it outputs a OutputDataObject. That object is then read by the Data Collector and stored in its internal database. The updated database is then sent to the interface as an update. The network administrator then closes the connection by sending a close command to the interface.

The following use cases assume that the interface is already open and not yet closed for simplicity.

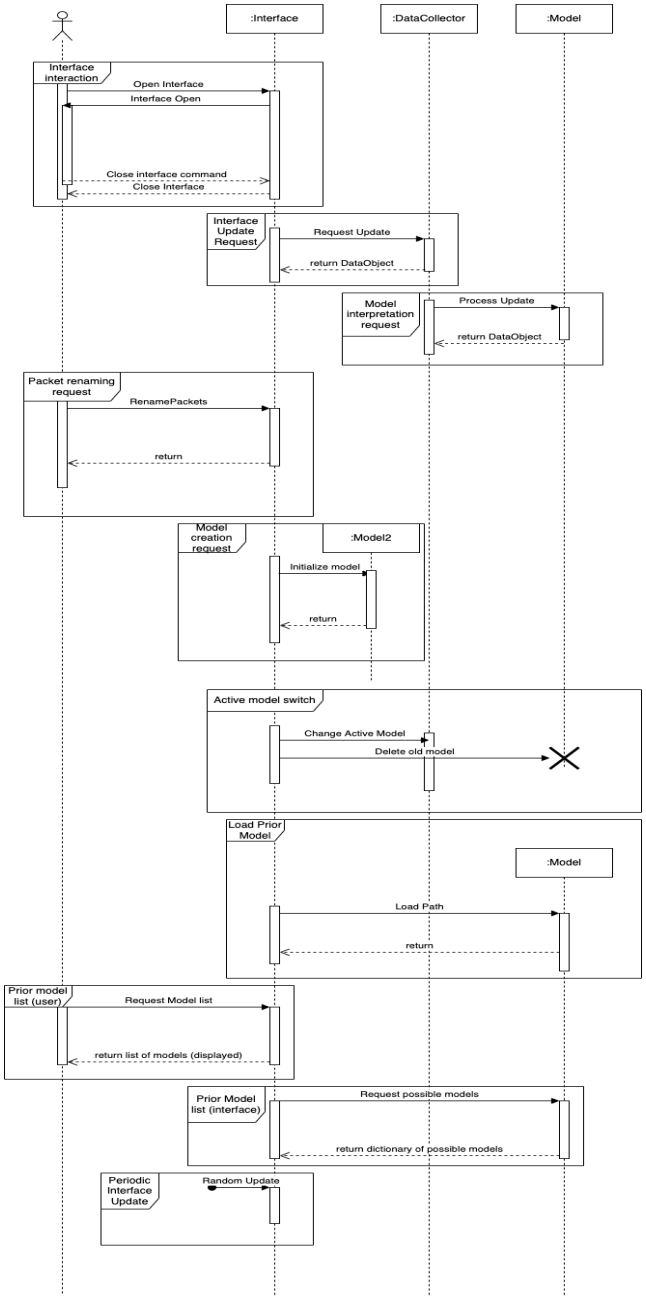
The second use case is to rename packets for training, these packets are unknown packets or packets identified as such by the model structure. This is a simple use case as the new name and packet information are just stored in a csv that the model can then read next time it trains.

The third use case is the model retraining. When the retrain, command is sent to the interface it creates a new model instance that begins reading the training data. This new model instance reads the starting training data as well as any additional training data that was created by relabeling unknown packets. Once the model has finished retraining it sends a confirmation to the interface that it has finished retraining. The interface then sends the new model information to the Data Collector. The Data Collector will now process data with the new model. Once that is complete, a command is sent to the old model to shut down.

Loading works similarly, except that instead of training, the model simply loads a previously trained network out of a file.

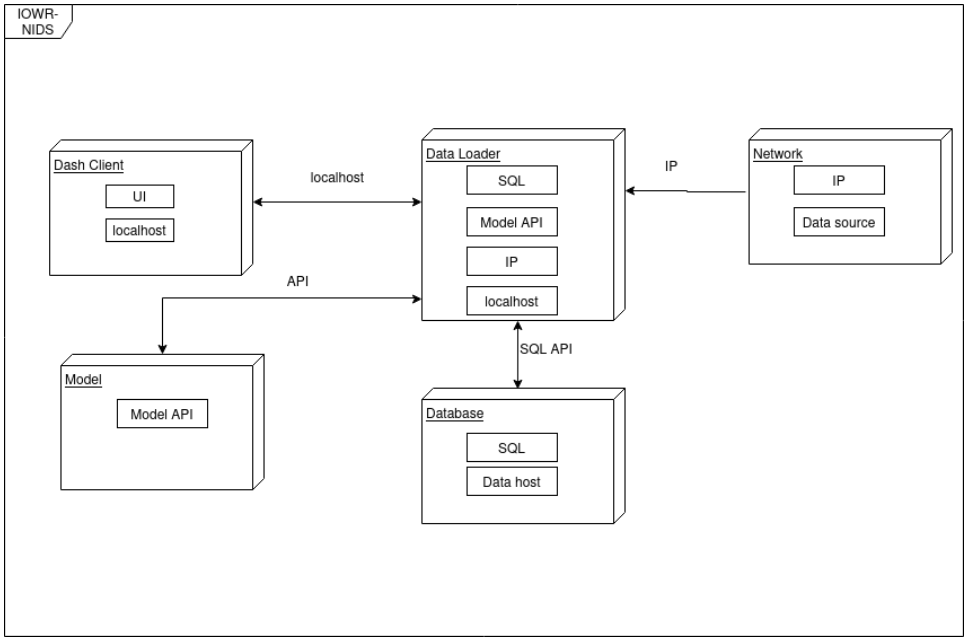
And finally, the interface should periodically poll the Data Collector for new updates even without the input of the Network Administrator.

6. Process View

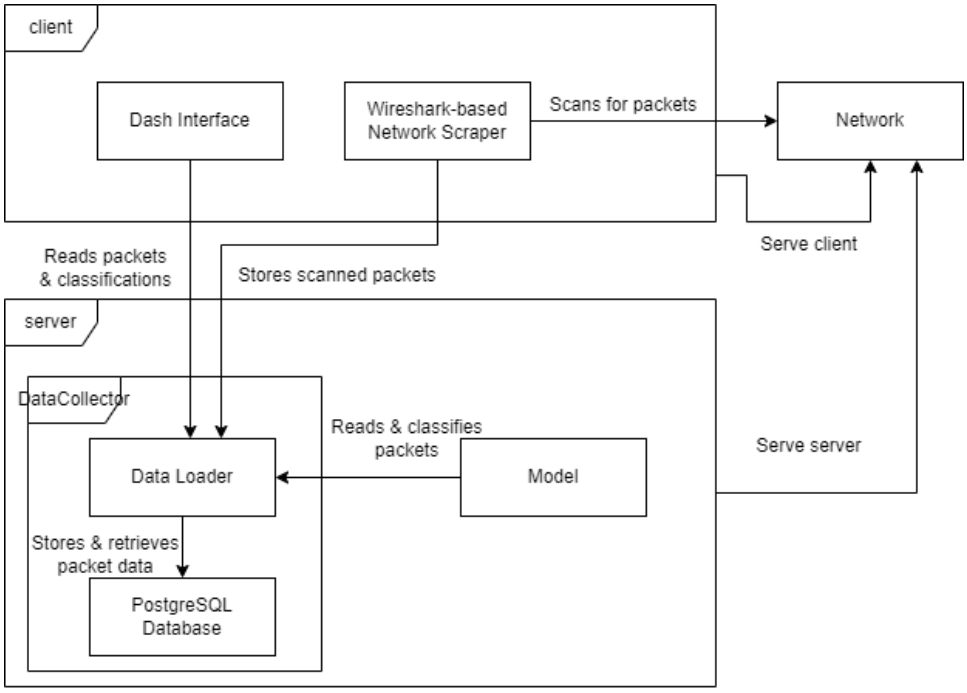


This is a breakdown of the individual lightweight process flows of each of the use cases seen in section 5.3. Combining these processes can create the use cases, which would be heavyweight processes. Communication within each process is done by message passing.

7. Deployment View



8. Implementation View



8.1 Overview

- PostgreSQL Database
  - PostgreSQL is a high-performance open-source database that will enable the user interface's rapid retrieval of packets. It works with the data loader, providing real-time information.
- Dash Interface

    - A web server based on Dash will provide metrics and configuration that interacts with the data loader.
- Data Loader
    - The data loader interfaces between the PostgreSQL database and everything else. A library such as Pyro, which allows us to communicate between processes, may be used to accomplish this.
- Network Scraper
    - The network scraper reads packets off the network and sends them back over the network, into the data loader.
- Model
    - The model will be used to read and classify packets that were found by the data loader.

8.2 Layers

For an idea as to the component hierarchy, observe the implementation diagram and the order of frames.

1. Client
    1.1. Dash Interface
    1.2. Wireshark-based Network Scraper
2. Server
    2.1. Data Collector
        2.1.1. Data Loader
        2.1.2. PostgreSQL Database
    2.2. Model
3. Network

9. Data View (optional)

The system will be using a PostgreSQL database to store individual packets and the information about those packets including source IP, destination IP, transfer protocol, assumed label, and assigned packet label.

The Data Collection module will be the only module directly interfacing with the PostgreSQL database and will translate this data into two different forms. One data form for the interface module and the other for the model module.

10. Size and Performance

The software must be as close to real-time as possible. With security, it is important to have the latest information. The server will send network data through the model and into the database; the client will poll for this data at a reasonably high rate (maybe once every 5 seconds).

The software must be scalable enough to handle large magnitudes of packets with minimal degradation. Using a trusted database like PostgreSQL enables this, as it likely finds this order of packets trivial.

11. Quality

The data collector allows for the database (or other) to be swapped out in situations where a different solution is necessary. As a middleman for all components, it will enable some level of extensibility.

Given the model, data collector, and database on a remote server, there is a higher level of reliability. The server model also pushes portability, as the client doesn’t need to be a heavyweight when it's time to retrain the model.