Project Report: Predictive Network Link Failure Analysis System

Abstract

This report details the design, implementation, and deployment of an end-to-end system for predicting network link failures using machine learning. The project encompasses four key stages: (1) dynamic data generation using the ns-3 network simulator to create a robust dataset of network performance metrics; (2) training and evaluation of a Random Forest classification model to distinguish between 'OK' and 'FAILURE' states; (3) deployment of the trained model via a Python-based Flask REST API; and (4) creation of an interactive web-based user interface for real-time predictions. The resulting system is a fully functional prototype demonstrating a practical application of machine learning for predictive network maintenance.

1. Introduction

The proactive identification of potential network link failures is a critical challenge in network management. Traditional monitoring systems are often reactive, raising alarms only after a failure has occurred, leading to downtime and service disruption. This project addresses this challenge by developing a predictive system capable of forecasting the status of a network link based on its real-time performance metrics. By leveraging a machine learning model, the system can identify subtle patterns in network behavior that precede a failure, enabling preemptive action.

The primary objective is to build a complete pipeline, starting from data synthesis and culminating in a user-facing application that provides actionable intelligence.

2. System Architecture

The project is architected as a four-stage pipeline, where the output of each stage serves as the input for the next. This modular design ensures a clear separation of concerns and facilitates independent development and testing of each component.

1. **Data Generation (tcp-realtime-logger1.cc):** An ns-3 simulation generates a rich dataset by modeling a point-to-point network link under various conditions. It logs key performance indicators (KPIs) like throughput and delay to a CSV file.
2. **Model Training (predict4.py):** A Python script consumes the generated CSV data. It preprocesses the data, trains a Random Forest classifier, evaluates its performance using rigorous metrics, and serializes the trained model into a .joblib file for deployment.
3. **Backend API (app.py):** A Flask web server loads the serialized model and exposes a /predict REST API endpoint. This backend service acts as the brain of

the operation, serving predictions on demand.

4. **Frontend Interface (cnp.html):** A standard HTML and JavaScript web page provides a user-friendly form. It captures user input for network metrics, communicates with the backend API, and displays the model's prediction in an intuitive, color-coded format.

3. Component Analysis

This section provides a detailed analysis of each of the four core components of the project.

3.1 Data Generation: ns-3 Simulation (tcp-realtime-logger1.cc)

This C++ code acts as a virtual network laboratory, using the ns-3 simulator to generate the raw data required for model training. Its primary purpose is to create a varied and realistic dataset of network performance under different conditions. The script establishes a simple two-node, point-to-point network and introduces randomization to ensure data diversity. For each simulation run, it randomly selects one of five different TCP congestion control algorithms (e.g., TcpNewReno, TcpVegas), which handle network congestion in unique ways. It also randomizes link delay, traffic start/stop times, and packet sizes.

The core of the script is the LogMetrics function, which runs every second of the simulation to capture instantaneous throughput (Mbps) and average packet delay (s). The most intelligent feature is its dynamic labeling logic. Instead of using fixed failure thresholds, it classifies the link status as FAILURE if the current throughput drops into the bottom 25% of historical values or if the delay rises into the top 75%. This context-aware method produces realistic labels that reflect relative performance degradation. The final output is a timestamped CSV file containing the data needed for the next stage.

3.2 Model Training and Evaluation (predict4.py)

This Python script is responsible for transforming the raw data from the ns-3 simulation into a trained, intelligent predictive model. It begins by finding and consolidating all generated CSV files into a single pandas DataFrame. The data is then prepared for training by removing any rows with missing values and encoding the text labels 'OK' and 'FAILURE' into numerical format (1 and 0).

A RandomForestClassifier is selected as the model. This ensemble method is robust, handles complex relationships in the data well, and is less prone to overfitting. A critical parameter, class_weight='balanced', is used to ensure the model gives

appropriate importance to the less frequent but more critical 'FAILURE' cases. The script rigorously evaluates the model by splitting the data into training (80%) and testing (20%) sets, performing 5-fold cross-validation for a stable accuracy metric, and generating a detailed Classification Report and Confusion Matrix. These diagnostics provide deep insight into the model's real-world performance. Once trained and validated, the final model is serialized and saved to a single file, link_failure_classifier.joblib.

3.3 Backend API Deployment (app.py)

This Python script serves as the bridge between the trained model and the end-user, making its predictive power accessible over the network. Using the Flask web framework, it creates a lightweight web server that loads the link_failure_classifier.joblib file into memory upon startup.

The script's main function is to expose a REST API endpoint at the URL /predict. This endpoint listens for POST requests, which are sent by the frontend interface. When a request is received containing throughput and delay data, the server formats this input into a pandas DataFrame that matches the structure the model was trained on. It then calls the model to get both a prediction ('OK' or 'FAILURE') and a confidence score for that prediction. The result is packaged into a JSON object and sent back as the response. The script also enables Cross-Origin Resource Sharing (CORS), a necessary security feature that allows the separately hosted HTML page to communicate with the server.

3.4 Frontend User Interface (cnp.html)

This HTML file is the sole interactive component of the project, providing a clean and intuitive interface for the user. The document is structured with standard HTML and styled with Tailwind CSS for a modern, responsive design. All interactive logic is handled by client-side JavaScript within the <script> tag.

The interface presents a simple form for entering Throughput and Delay values. An event listener waits for the user to click the "Predict Status" button. Upon clicking, an asynchronous JavaScript function is triggered, which uses the browser's fetch API to send the form data to the backend Flask server's /predict endpoint. While waiting for a response, the button provides feedback that it is "Predicting...". Once the JSON response is received from the server, the script dynamically updates the page to display the prediction and confidence score. The result box changes color—green for 'OK' and red for 'FAILURE'—providing immediate visual feedback. The script also includes robust error handling to inform the user if it cannot connect to the backend

server.

4. Conclusion

This project successfully demonstrates the creation of a complete, end-to-end machine learning system for a real-world problem. By integrating network simulation, model training, API deployment, and a web frontend, it provides a powerful proof-of-concept for predictive network failure analysis. The modular architecture allows for each component to be independently improved, such as by training the model on real network data or deploying the web application to the cloud. The final result is a practical and interactive tool that effectively translates complex network data into a simple, actionable prediction.

Of course. Your project cleverly uses several network protocols across its different stages, from simulation to the live application. Let's break them down by their layer in the network stack.

Application Layer Protocols

These protocols define how applications interact with each other.

1. **HTTP (Hypertext Transfer Protocol)**
    - **What it is:** The backbone of the World Wide Web. It's a request-response protocol where a client (like a web browser) requests data, and a server responds with it.
    - **Where it's used:** In the communication between your **web page (`live_dashboard.html`)** and your **backend server (`app.py`)**.
    - **Why it's used:** When you enter data and click "Predict Status" on the web page, the JavaScript `fetch` command sends an HTTP `POST` request to your Flask server. The Flask server processes this request, runs the model, and sends the prediction back in an HTTP response. The `/stats` endpoint uses an HTTP `GET` request. It's the standard, universal way for web frontends to talk to backend APIs.
2. **(Simulated) OnOff Application Protocol**
    - **What it is:** This isn't a standard internet protocol but a **simulated application** provided by ns-3. The `OnOffHelper` creates a traffic source that alternates between "On" (sending data at a constant rate) and "Off" (sending nothing).
    - **Where it's used:** Inside the **ns-3 simulation (`tcp-realtime-logger1.cc`)**.

- **Why it's used:** Its purpose is to generate a predictable but varied stream of data to send over the simulated TCP connection. This allows you to create realistic network load and measure how the link's performance (throughput, delay) changes in response, which is essential for creating good training data.

Transport Layer Protocols

This layer is responsible for end-to-end communication and data integrity.

1. **TCP (Transmission Control Protocol)**
   - **What it is:** A reliable, connection-oriented protocol. It ensures that all data sent from one point arrives at the other end, intact and in the correct order. It manages this with features like acknowledgments, retransmissions for lost packets, and congestion control.
   - **Where it's used:**
     1. **Explicitly in the ns-3 simulation:** The entire simulation is built to model and measure the performance of different TCP variants (`TcpNewReno`, `TcpVegas`, `TcpWestwood`, etc.). The goal is to see how these algorithms perform under different conditions.
     2. **Implicitly in the live application:** HTTP, the protocol used by your Flask app and frontend, runs on top of TCP. It relies on TCP's reliability to ensure the API requests and responses are not corrupted.
   - **Why it's used:** It's the core subject of your simulation and the standard for most reliable web communication, making it the perfect protocol to model for predicting general link failures.

Internet Layer Protocols

This layer handles addressing and routing packets across networks.

1. **IP (Internet Protocol - specifically IPv4)**
   - **What it is:** The principal protocol for routing traffic across the internet. Its main job is to assign unique addresses (IP addresses) to devices and route packets of data from a source to a destination based on these addresses.
   - **Where it's used:**
     1. **In the ns-3 simulation:** The `Ipv4AddressHelper` is used to assign IP addresses (like `10.1.1.1`, `10.1.1.2`) to the simulated nodes.

2. **In the live application:** When your browser sends a request to `http://127.0.0.1:5000`, it's using the IP address `127.0.0.1` (the "localhost" address) to find the Flask server running on your own machine.
    - **Why it's used:** It is the fundamental protocol for all internet communication. You cannot have a TCP/IP network without it.

Data Link Layer Protocols

This layer handles communication between devices on the *same* local network segment.

1. **Point-to-Point Protocol (PPP)**
    - **What it is:** A protocol for establishing a direct connection between two network nodes.
    - **Where it's used:** The `PointToPointHelper` in your **ns-3 simulation** is used to create the fundamental link between your two nodes.
    - **Why it's used:** It provides a simple, clean, and controllable way to model a direct network link, allowing you to precisely set its characteristics (like data rate and delay) and measure its performance without the complexity of other network types like Wi-Fi or Ethernet.

The Role of TCP (Transmission Control Protocol)

Before we discuss the variants, it's important to understand TCP's fundamental job. It operates at the Transport Layer of the network stack and provides **reliable, ordered, and error-checked delivery of a stream of data**.

Think of it as a highly professional courier service, whereas its counterpart, UDP, is like standard mail. TCP guarantees delivery through several key mechanisms:

1. **Three-Way Handshake:** It establishes a reliable connection before any data is sent (`SYN`, `SYN-ACK`, `ACK`).
2. **Sequence Numbers:** It numbers every byte of data it sends. The receiver uses these numbers to reassemble the data in the correct order.
3. **Acknowledgments (ACKs):** The receiver sends acknowledgments back to the sender to confirm which data has been received successfully.
4. **Retransmission:** If the sender doesn't receive an ACK for a piece of data within a certain time (a timeout), it assumes the data was lost and sends it again.

The Core Problem: Congestion Control

The most complex and important part of TCP, and the focus of your simulation, is **congestion control**. The internet is a shared resource with finite capacity. If a sender sends data too fast, it can overwhelm the routers along the path, causing their internal buffers to overflow and packets to be dropped. This is network congestion.

A TCP congestion control algorithm is a set of rules that a sender uses to:

1. Probe the network to find out how much available capacity there is.
2. Adjust its sending rate to use that capacity efficiently without causing congestion.
3. React quickly when it detects that congestion (i.e., packet loss) is occurring.

All the variants you simulated share a common goal but use different strategies to achieve it. The main phases of operation are:

- **Slow Start:** At the beginning of a connection, the sender starts by sending a small number of packets (the congestion window, or `cwnd`) and rapidly doubles it for every ACK received. This allows it to quickly discover the available bandwidth.
- **Congestion Avoidance:** Once the congestion window reaches a certain threshold (`ssthresh`), the sender slows down its rate of increase, growing the window additively (e.g., by one packet per round-trip time) instead of exponentially.
- **Packet Loss Detection:** The algorithm detects packet loss in two ways:
    - **Timeout:** The sender gets no ACK for a packet at all. This is a severe sign of congestion.
    - **Triple Duplicate ACKs:** The sender receives three ACKs in a row for the same packet, which implies that the *next* packet was lost but that the network is otherwise still flowing.

Now let's look at the specific strategies of the variants you used.

The TCP Variants in Your Project

1. TcpTahoe (The Classic)

- **Strategy:** Aggressive and simple. It was one of the first algorithms to combine Slow Start and Congestion Avoidance.
- **How it Reacts to Loss:**
    - **On Triple Duplicate ACK:** It performs a "fast retransmit" of the lost

packet, but then it immediately **resets its congestion window to 1** and enters Slow Start again. It also cuts its `ssthresh` in half.
    - **On Timeout:** Same as above. It resets `cwnd` to 1 and enters Slow Start.
- **Implication:** Tahoe is very cautious. Any sign of packet loss makes it slam on the brakes and start over slowly. This is safe but can be very inefficient, causing the "sawtooth" pattern of throughput to have very sharp, deep drops.

2. TcpReno (The Improvement)

- **Strategy:** A refinement of Tahoe that is less aggressive on minor packet loss.
- **How it Reacts to Loss:**
    - **On Triple Duplicate ACK:** This is its key innovation. It performs a fast retransmit, but instead of resetting `cwnd` to 1, it **halves the** `cwnd` and enters a new phase called **Fast Recovery**. It stays in this phase, artificially inflating the window based on further duplicate ACKs, until it gets an ACK for the recovered data. This avoids the drastic throughput drop of Slow Start.
    - **On Timeout:** It behaves just like Tahoe, resetting `cwnd` to 1 and entering Slow Start.
- **Implication:** Reno is the foundation for most modern TCPs. Its ability to handle single packet losses without a full reset makes it much more efficient than Tahoe.

3. TcpNewReno (The Refinement)

- **Strategy:** An improvement on Reno's Fast Recovery mechanism.
- **The Problem it Solves:** Classic Reno exits Fast Recovery as soon as it gets an ACK for the first retransmitted packet. If *multiple* packets were lost from the same window of data, Reno would immediately get another triple duplicate ACK and halve its window again, leading to poor performance.
- **How it Reacts to Loss:** NewReno's Fast Recovery is "smarter." When it receives an ACK during recovery, it checks if it's a "partial ACK" (meaning it acknowledges some but not all of the data that was in flight when recovery started). If it is, NewReno assumes another packet was lost and immediately retransmits it *without* exiting Fast Recovery. It only exits once all data from that original window is acknowledged.
- **Implication:** NewReno is significantly more robust and efficient than Reno when multiple packets are dropped in a short period, which is common in real networks.

4. TcpVegas (The Proactive One)

- **Strategy:** Fundamentally different from the others. Vegas is a **proactive**, delay-based algorithm. Instead of reacting to packet loss, it tries to *prevent* it from happening in the first place.
- **How it Works:** It constantly measures the Round-Trip Time (RTT). It compares the current RTT to the minimum RTT it has ever seen. The difference between these two tells it how much "extra" data is currently queued up in the network's router buffers.
    - If the queue size is too small, Vegas increases its sending rate.
    - If the queue size is too large (indicating buffers are filling up and congestion is imminent), it *decreases* its sending rate.
- **Implication:** Vegas aims for higher throughput with fewer packet losses and a smoother sending rate. However, when competing with loss-based algorithms like Reno on the same link, it can be too "polite," backing off its sending rate while Reno continues to be aggressive, causing Vegas to get a smaller share of the bandwidth.

5. TcpWestwood (The Smart Estimator)

- **Strategy:** A sender-side-only modification of TCP Reno that is particularly effective on networks with high packet loss that isn't caused by congestion (like wireless or satellite links).
- **How it Works:** Its key innovation is **bandwidth estimation**. When a packet loss occurs (either by triple duplicate ACK or timeout), a standard algorithm like Reno blindly halves its congestion window. TcpWestwood does something smarter. It continuously monitors the rate of returning ACKs to estimate the actual bandwidth currently available on the connection.
- **How it Reacts to Loss:** After a packet loss, instead of just halving `cwnd`, it sets its `cwnd` and `ssthresh` to a value based on its **bandwidth estimate**. The idea is to set the window to a size that the network can actually support, rather than an arbitrary smaller number.
- **Implication:** This makes Westwood much more efficient and fair, especially on lossy links. It can recover from packet loss much faster and more accurately than Reno, leading to better overall throughput.