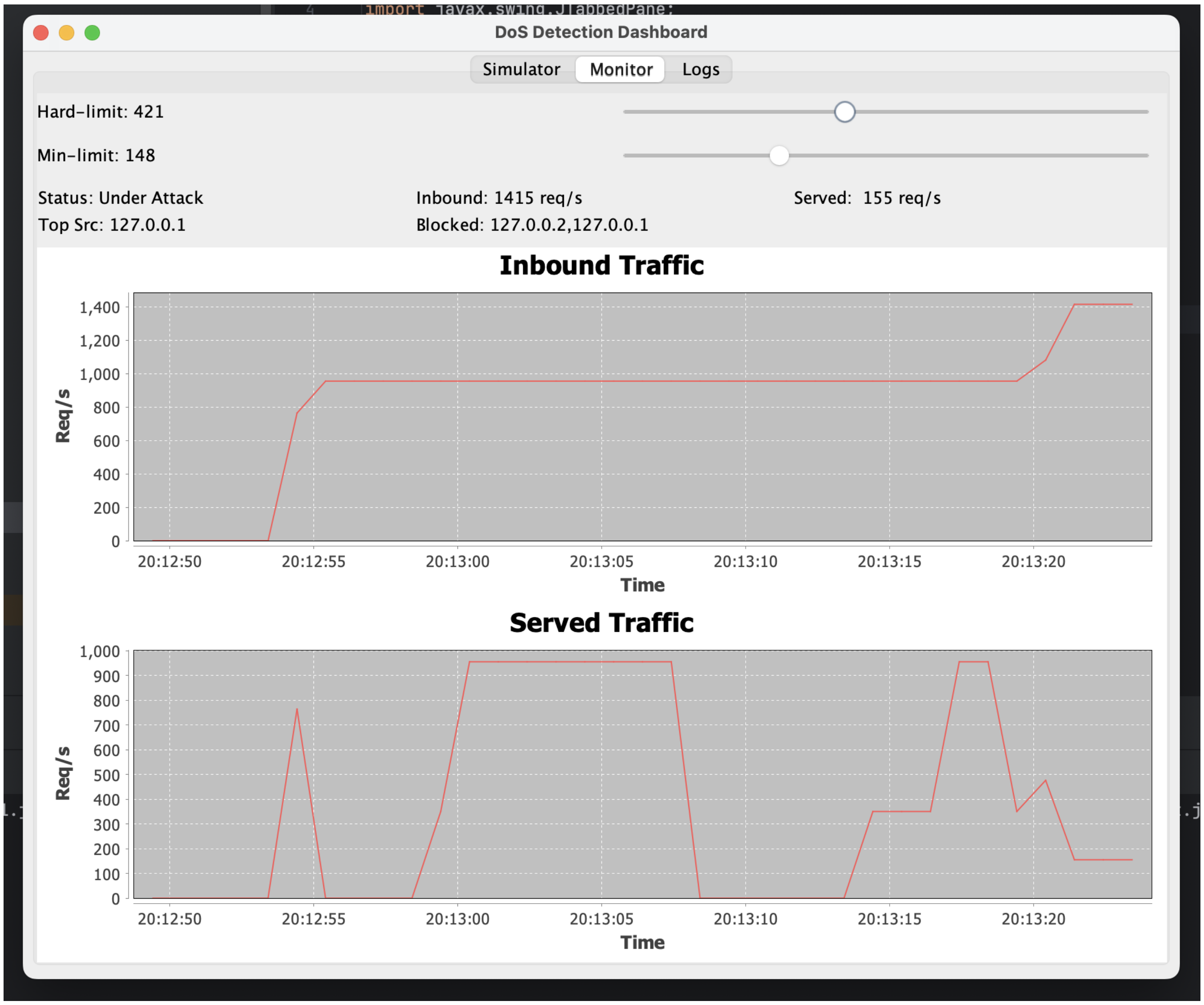
(Simple) DoS Detection Project

Matic Bončina
89231371@student.upr.si



Hard-limit: 421

Min-limit: 148

Status: Under Attack
Inbound: 1415 req/s
Served: 155 req/s
Top Src: 127.0.0.1
Blocked: 127.0.0.2,127.0.0.1

Figure 1: GUI of the monitoring dashboard

ABSTRACT

This document describes a proof-of-concept implementation of a lightweight, non-production DoS detection system with automated null-routing developed in Java. We examine two deployment options, parallel execution on a single node and distributed processing using MPJ Express, to see which one results in higher detection throughput and shorter response latency. Purely sequential processing is infeasible because it causes unreasonable packet buffering delays and connection dropouts during attacks. As a result, we focus solely on parallel and distributed setups. The prototype is divided into two modules (parallel and distributed) and is accessible on GitHub (https://github.com/maticboncina/famnit24-prog3-projekt). Readers should read the README for extensive usage instructions, constraints, and design issues before evaluating.

Programming 3, July 2025, FAMNIT
2025.

KEYWORDS

Denial-of-Service (DoS) Detection, Automatic Null-Routing, Parallel Processing, Distributed Processing, MPJ Express, Java, Throughput Optimization, Latency Reduction, Proof-of-Concept Implementation, Network Security Tools

ACM Reference Format:
Matic Bončina. 2025. (Simple) DoS Detection Project. In *Proceedings of Concurrent Programming Course Project (Programming 3).* ACM, New York, NY, USA, 4 pages.

1 INTRODUCTION

Denial of Service (DoS) is a well-known attack vector aimed at consuming a server's resources to prevent legitimate clients from accessing services. From simple TCP floods to more advanced, application-level assaults, DoS attacks have become increasingly sophisticated as providers implement deeper protection layers. In this project, we monitor live traffic by invoking an external packet capture tool (e.g., tcpdump) via Java's ProcessBuilder; the TCP packet stream is treated as a producer, with multiple consumer threads analyzing the output in real time.

Our goal is to build a basic web server (or SocketServer) and simulate both normal and malicious traffic using Apache Benchmark (ab). We will start by implementing a single-threaded anomaly detector to establish baseline performance, then refactor the code into a multithreaded design for higher throughput. Finally, we will explore a distributed version using MPJ Express to run the detector across multiple JVM instances. Anomaly detection leverages statistical indicators—such as moving averages and volatility metrics like Bollinger Bands—to flag statistically significant traffic spikes on a per-source basis. To complement the core functionality, we will develop graphical interfaces: a client-side panel to simulate traffic patterns (including DoS attacks) and a server-side dashboard to display throughput, request counts over time, and real-time attack status.

2 REQUIREMENTS

The system must satisfy the following requirements as specified in the project instructions and description:

(1) Invoke an external packet-capture tool (`tcpdump`) via Java's `ProcessBuilder`, capturing traffic on a specified port and streaming its output into the application for processing .
(2) Spawn consumer threads that read from the `ProcessBuilder`'s `InputStream` and perform real-time traffic monitoring by parsing each TCP packet line .
(3) Implement a basic `WebServer` or `SocketServer` in Java to serve as the target for simulated traffic and to integrate with the detection logic.
(4) Simulate both normal and attack-level traffic using external benchmarking tools (e.g., Apache Benchmark `ab`) against the server to test the detection mechanism.
(5) Incorporate anomaly detection based on statistical indicators—such as moving averages and volatility measures (e.g., Bollinger Bands)—to flag significant traffic spikes per source IP.

3 STRUCTURE

All development and testing were carried out on a personal Mac running Apple Silicon M1 Pro and macOS, with the entire system implemented in Java. Both the sequential detector and the distributed version (using MPJ Express) run natively on the Mac without. To streamline deployment and management, the traffic simulator (client) and the anomaly detector (server) are packaged together in a single Java application, exposing a unified GUI with tabs for "Simulator," "Monitor," and "Logs".

4 GUI

The graphical interface is organized into three main views—"Simulator," "Monitor," and "Logs"—all sharing the same overall layout and controls. The only difference between sequential and distributed modes lies in how the backend dispatches tasks (single JVM vs. MPJ Express).

4.1 Simulator

Implemented by the `AttackSimulatorPanel`, this view allows the user to configure and launch traffic generation against the target server

- **Target URL:** A text field for the server endpoint (e.g. `http://127.0.0.1:80`
- **Client Configuration:** One row per simulated client IP (e.g. 127.0.0.1, 127.0.0.2, 127.0.0.3), each with:
  - A checkbox to enable/disable that client.
  - A slider to set request rate (requests per second).
- **Controls:** "Start Simulation" and "Stop Simulation" buttons manage a pool of scheduled tasks that issue HTTP requests with the configured `X-Forwarded-For` header.

4.2 Monitor

Implemented by the `MonitoringPanel`, this view provides real-time traffic metrics and anomaly detection status:

- **Limits Panel:** Two sliders to adjust the "hard" and "min" RPS thresholds used in detection.
- **Info Panel:** Labels showing current status (`Normal` or `Under Attack`), inbound and served RPS, top source IP, and list of currently blocked IPs.
- **Charts:** Two time-series plots (JFreeChart) displaying inbound and served request rates over a sliding window.

4.3 Logs

Implemented by the `FirewallLogPanel`, this view displays timestamped events (e.g. "Blocked 127.0.0.2 (rps=450)") in a scrollable text area as IPs are blocked or unblocked based on the detection logic.

5 SEQUENTIAL VERSION

The sequential implementation runs entirely within a single JVM and consists of three main phases:

(1) **Packet capture.** At startup, we invoke `tcpdump` via Java's `ProcessBuilder`:

```
sudo tcpdump -l -n -i any port 8080 and (tcp-syn|tcp-ack)!=0
```

Its output is redirected into a `LinkedBlockingQueue<String>` for processing.

(2) **Packet consumption & anomaly detection.** Two threads consume the queued lines:
- A "reader" thread continuously reads from `tcpdump`'s `InputStream` and enqueues each packet line.
- A "consumer" thread dequeues lines, parses out the source IP, and updates global (`totalAttempted`) and per-IP (`attemptsByIp`) counters.

Every second, a scheduled task computes the per-IP request delta, updates a sliding-window statistics object (`TrafficStats`) to calculate mean and standard deviation (used to form Bollinger Bands), and applies thresholds to block or unblock IPs.

(3) **HTTP server & traffic simulation.** We embed a basic `HttpServer` on port 8080:
- Incoming requests increment `totalAttempted` and, if not blocked, `totalServed`.
- Responses are a simple HTML page.

In parallel, the `AttackSimulatorPanel` GUI uses a `ScheduledExecutorService` to issue HTTP GETs (with `X-Forwarded-For` headers) at a user-configured rate per client IP, controlled via sliders in the Simulator tab.

All components—packet capture, detection, HTTP serving and simulation—are wired up in `TCPDumpServer.main()`, and the unified Swing GUI presents three tabs ("Simulator," "Monitor," "Logs") for user interaction and real-time feedback.

6 DETECTION METHODS

The system supports two complementary modes for identifying anomalous traffic patterns:

6.1 Automatic (statistical) detection

Traffic for each source IP is recorded in a sliding window (up to 60 measurements). On each interval:

- Compute the mean and standard deviation of requests/sec over the window.
- Form an upper bound via Bollinger Bands (mean + 2×stddev).
- If the current rate exceeds both the minimum activity threshold and either the hard limit or the Bollinger upper bound, the IP is blocked.
- IPs are unblocked only after three consecutive intervals below both thresholds.

6.2 Manual (fixed thresholds) detection

Two user-adjustable sliders control:

- `hardLimit`: a static requests/sec ceiling above which any IP is immediately blocked.
- `minLimit`: a minimum activity threshold; rates below this are never considered for blocking regardless of spikes.

These sliders allow an operator to override or fine-tune automatic behavior in real time via the Monitor tab :contentReference[oaicite:13]index=13

7 PARALLEL VERSION

The distributed MPJ version is implemented using a master–worker MPI topology. The master process reads 100 synthetic HTTP packets (simulating traffic to port 8080), timestamps each packet, and then distributes them in a round-robin fashion to four worker processes via non-blocking MPI sends. Each worker, upon receipt, performs parsing, IP-extraction and a 0.5 ms simulated security-analysis delay before returning the result to the master with an MPI send. High-resolution timing (via `System.nanoTime()`) captures end-to-end latency per packet. Load balancing is ensured by the master's round-robin scheduler, and MPI's inherent thread safety removes critical-section concerns inside each worker.

7.1 Baseline single-threaded testing

To establish a reference, the same 100 packets were processed end-to-end in a single JVM thread with no MPI involvement. Each packet incurs the fixed 0.5 ms processing delay and is handled sequentially. Total execution time, per-packet latency, and throughput (packets/sec) were measured once, yielding:

- **Total execution time**: 68.08 ms
- **Average per-packet delay**: 0.678 ms
- **Throughput**: 1 469 packets/sec

7.2 Parallel findings

Running the MPJ version on four cores produced dramatic gains:

- **Total execution time**: 19.03 ms (3.58× faster)
- **Average per-packet delay**: 0.660 ms (2.6
- **Throughput**: 5 255 packets/sec (257.7
- **Parallel efficiency**: 89.4

These results demonstrate that packet processing is an embarrassingly parallel workload with minimal MPI overhead, enabling real-time DoS detection on high-volume traffic.

8 FUTURE IMPROVEMENTS

At this stage the project is best regarded as a proof of concept rather than a production-ready solution. Many of the planned "improvements" outlined earlier—regex filters, headless mode, advanced offloading—feel like cosmetic add-ons compared to the real challenges of DoS detection. In reality, robust mitigation requires advanced, data-driven algorithms (e.g. machine-learning classifiers on NetFlow/SFlow data, BGP Flowspec rules, ISP-level volumetric filtering) rather than simple threshold or volatility rules. Future work should focus on integrating with carrier-grade scrubbing centers at T1/T2 providers, deploying edge detection modules in upstream PoPs, and leveraging real-time telemetry feeds for adaptive filtering. A full architectural refactor (to eliminate code duplication and improve modularity) remains necessary, but more importantly the project should adopt proven enterprise solutions and threat feeds rather than bespoke prototype heuristics.

9 CONCLUSION

This project has served as a valuable exercise in end-to-end DoS proof of concept—from packet capture to anomaly detection to traffic simulation. Implementing both sequential and distributed detectors in a single Java application taught me the practical challenges of real-time processing, multithreading, and GUI integration on Apple Silicon. However, my experience working in datacenter operations has shown that true DDoS mitigation is an arms race: it demands carrier-grade volumetric scrubbing, advanced machine-learning models trained on vast telemetry datasets, and coordinated filtering at the network edge. Despite my best efforts to craft a home-grown mitigation technique, the complexity and scale of modern attacks render such approaches largely academic. In production environments, evolving enterprise solutions with mature threat intelligence and adaptive filtering pipelines remain the only scalable defense against large-scale DoS and DDoS campaigns.

REFERENCES