Network Intrusion Detection System

Date: 4/29/2025

Name	Email Address
John Gahagan	john.gahagan398@topper.wku.edu
Freddy Goodwin	frederick.goodwin919@topper.wku.edu

CS 381
4/29/2025
Project Technical Documentation

Contents

1 Introduction **1**
1.1 Project Overview 1
1.2 Project Scope 1
1.3 Technical Requirements 1
1.3.1 Functional Requirements 1
1.3.2 Non-Functional Requirements 2
1.4 Target Hardware Details 2
1.5 Software Product Development 2

2 Modeling and Design **2**
2.1 Wireframes and Storyboard 2
2.1.1 Class Diagrams 3
2.1.2 Use Case Diagram 4
2.1.3 Sequence Diagrams 5
2.1.4 State Diagrams 6
2.1.5 Component Diagrams 7
2.1.6 Deployment Diagrams 7
2.2 Version Control 8
2.3 User Interface and User Experience (UI/UX) 8

3 Software Testing **9**
3.1 Testing Overview 9
3.1.1 Performance Testing 9
3.1.2 Unit Testing 9
3.1.3 Integration Testing 13
3.1.4 System Testing 14
3.1.5 Acceptance Testing 15
3.2 Packet Analysis with Wireshark 16

4 Conclusion **17**

5 Appendix **17**
5.1 Build Instructions 17
5.2 User Guide 17
5.3 Source Code Files 18

List of Figures

1 Storyboard: IDS process from detection to alert . . . 3
2 Class Diagram for N.I.D.S. . . . 4
3 Use Case Diagram for Network Intrusion Detection System . . . 5
4 Sequence Diagram for Threat Detection Process . . . 6
5 State Diagram for Network Intrusion Detection System . . . 7
6 Component Diagram for Network Intrusion Detection System . . . 7
7 Deployment Diagram for Network Intrusion Detection System . . . 8
8 Test file ensuring that AlertSender.java can properly send UDP packets . . . 10
9 Simplified Client Handler file made to test loadThreatSignatures() shown printing out the contents of ThreatSignatures.txt . . . 11
10 Simplified Client Handler file made to test isThreat() shown printing out the contents of ThreatSignatures.txt and the result of isThreat(”unauthorized-access”) . . . 12
11 Simplified Server and Client files to verify TCP connectivity . . . 13
12 Client file and altered AlertSender file to verify UDP connectivity . . . 14
13 Firebase Realtime Database showing logged threat events . . . 15
14 Filtered Wireshark Capture for IDS TCP and UDP Communication . . . 16

1 Introduction

1.1 Project Overview

This project implements a Network Intrusion Detection System (NIDS) using Java, TCP/UDP socket programming, and Firebase. It monitors real-time network traffic, detects threats via pattern matching, sends alerts through UDP, and logs the activity. The system addresses the growing need for automated, real-time network monitoring tools for administrators and security teams.

It includes:

- Packet capturing using TCP sockets.
- Signature-based threat detection.
- Real-time alerts using UDP.
- Logging of incidents into Firebase.

1.2 Project Scope

Deliverables:

- A Java-based NIDS with a server and client component.
- A Firebase-integrated logging system.
- Alert system through UDP messages.
- Technical documentation with diagrams and testing results.

1.3 Technical Requirements

Our project satisfies all mandatory technical requirements outlined for CS 381 and exceeds the minimum by implementing seven total attributes. Socket programming is used through TCP sockets in `IDS_Server.java` and `IDS_Client.java` for client-server communication, and UDP sockets through `AlertSender.java` for real-time alert delivery. Process-to-process communication is established between the independently running IDS server and client programs, which exchange network data over TCP and UDP connections. The system maintains at least three active sockets during runtime, with TCP connections handling client packet transmissions and UDP sockets transmitting real-time threat alerts. Input is gathered manually from the keyboard in the client terminal and from the external `ThreatSignatures.txt` file, loaded inside `ClientHandler.java`. Detected threat events are output to a Firebase Realtime Database using `FirebaseLogger.java`, where relevant details like IP address, threat message, and timestamp are recorded. The system architecture follows a classic client-server model, where the server handles detection and logging, and the client handles packet simulation and alert reception. Finally, the system addresses network security by detecting known malicious packet patterns and immediately notifying administrators through UDP-based alerts.

1.3.1 Functional Requirements

Mandatory Functional Requirements
Socket Programming
Process-to-Process Communication
At least Three TCP/UDP Sockets
Input from Database, Keyboard, or File
Output to Database
Client/Server Architecture

Extended Functional Requirements
Network Security

1.3.2 Non-Functional Requirements

Mandatory Non-Functional Requirements
System must be intuitive and easy to use
Must be secure (data encryption, secure access)
Must support reconnection attempts
Extended Non-Functional Requirements
System must be scalable to multi-sensor environments
Optimized performance and resource usage
Cross-platform support

1.4 Target Hardware Details

- CPU: x86_64 architecture (Intel i5/Ryzen 5 or better)
- RAM: 4GB minimum
- Storage: 500MB available
- Network: Ethernet or Wi-Fi (stable connection required)
- Input: Keyboard and optional file-based commands
- Output: Monitor (1080p), Firebase database

1.5 Software Product Development

- **Language:** Java
- **Database:** Firebase Realtime Database
- **Version Control:** GitHub
- **IDE:** IntelliJ / Eclipse
- **Collab Tools:** GitHub Projects, Discord

2 Modeling and Design

2.1 Wireframes and Storyboard

The client UI includes a real-time alert dashboard and log viewer. The storyboard follows this path:

1. The IDS Server and Client Programs are started.
2. The system begins scanning network traffic for threats.
3. Admin logs in.
4. Real-time alerts appear if threats are detected.
5. Admin views past logs and threat metadata.

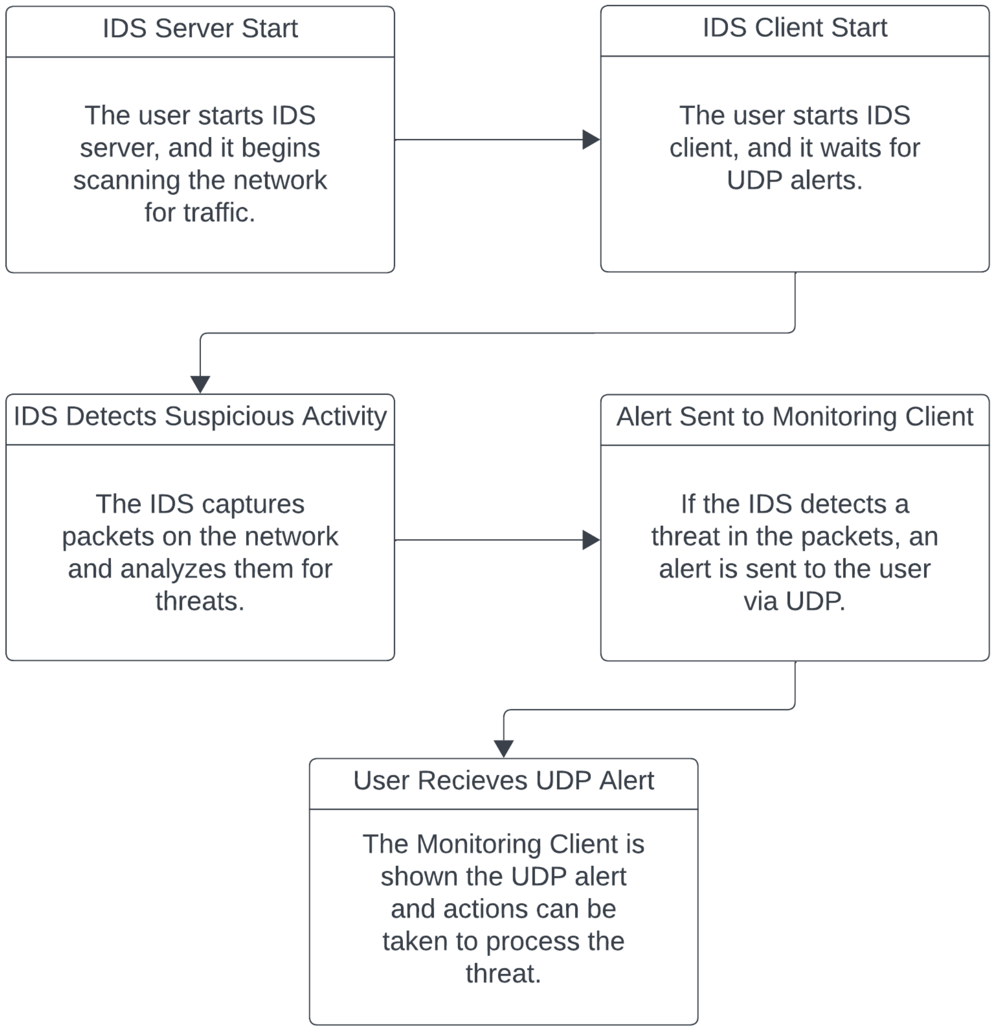

IDS Server Start
The user starts IDS server, and it begins scanning the network for traffic.
IDS Client Start
The user starts IDS client, and it waits for UDP alerts.
IDS Detects Suspicious Activity
The IDS captures packets on the network and analyzes them for threats.
Alert Sent to Monitoring Client
If the IDS detects a threat in the packets, an alert is sent to the user via UDP.
User Recieves UDP Alert
The Monitoring Client is shown the UDP alert and actions can be taken to process the threat.

Figure 1: Storyboard: IDS process from detection to alert

2.1.1 Class Diagrams

The class diagram below represents the main components of the N.I.D.S.

- **IDS_Server**: Listens on a socket for incoming network traffic and spawns new ClientHandler per connection.
- **ClientHandler**: Processes network traffic from a single client, scans for threat patterns, and coordinates alerts and logging.
- **AlertSender**: Sends out alerts over UDP when a threat is detected.
- **FirebaseLogger**: Connects to Firebase and logs detailed information about detected threats.

Each class has relevant methods for the role it has. The `ClientHandler` class depends on both `AlertSender` and `FirebaseLogger`, while `IDS_Server` manages multiple `ClientHandler` instances. These relationships are represented with associations in the diagram.

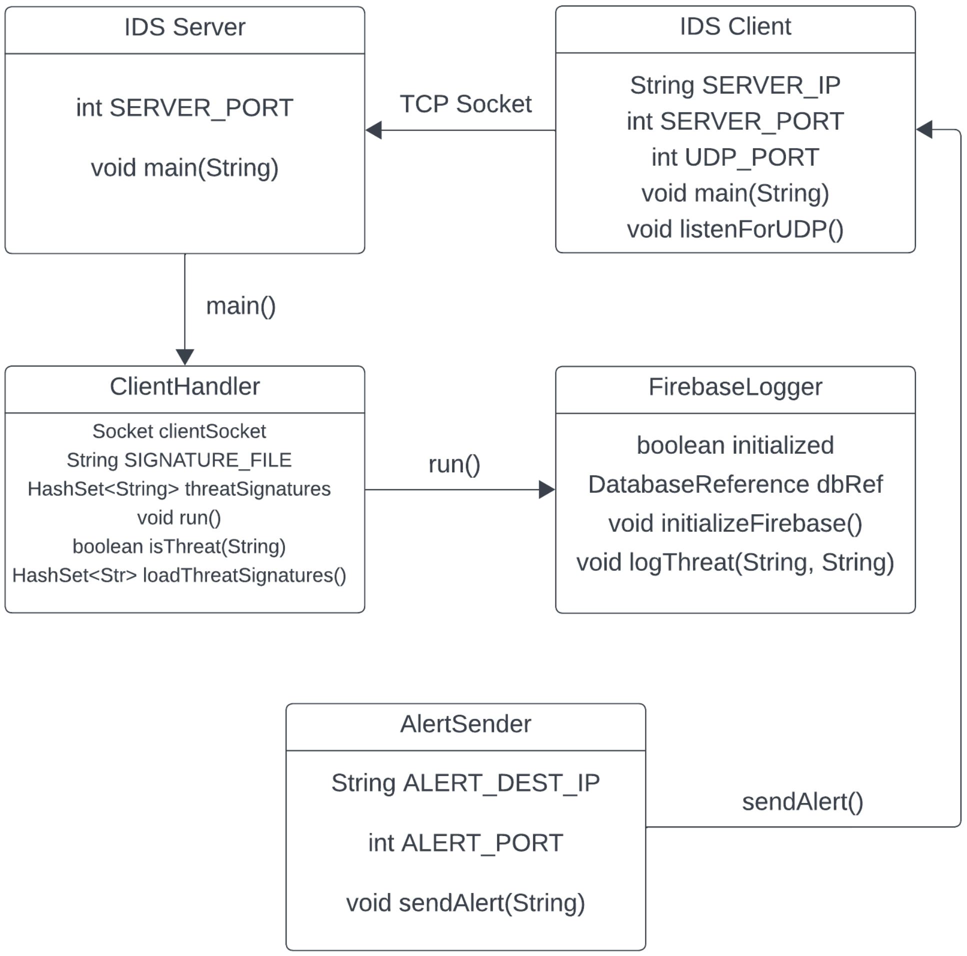


Figure 2: Class Diagram for N.I.D.S.

2.1.2 Use Case Diagram

- **Start System**: Admin launches the IDS server and client programs.
- **View Logs**: Admin checks previously logged alerts in Firebase.
- **Update Signatures**: Admin can push new threat patterns to the system (optional feature).
- **Monitor Network Traffic**: The system listens for incoming packets.
- **Detect Threat**: The system scans for known patterns.
- **Send Alert**: Sends a real-time UDP alert when a threat is found.
- **Log Incident**: Records the event and metadata to Firebase.

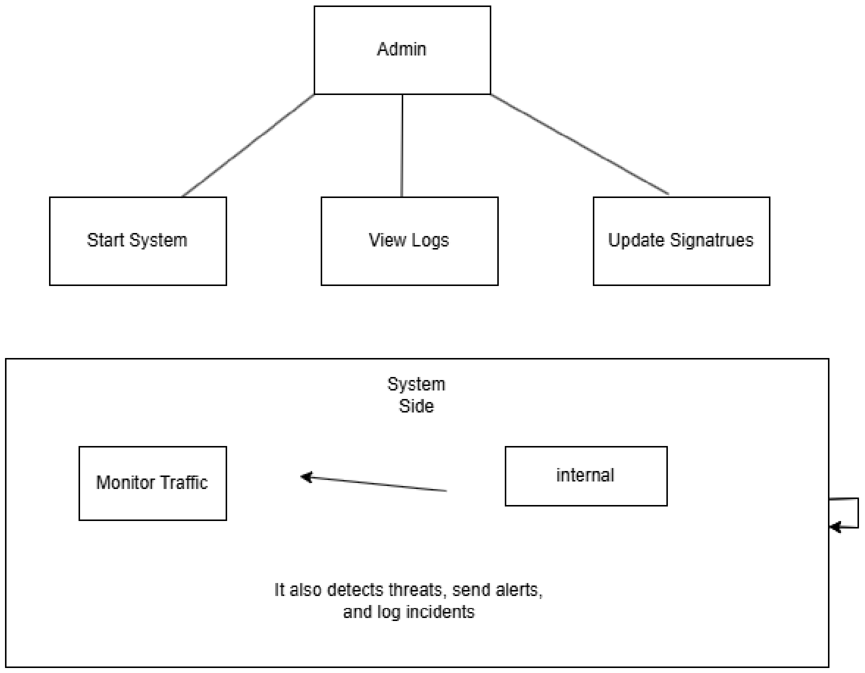


Figure 3: Use Case Diagram for Network Intrusion Detection System

2.1.3 Sequence Diagrams

The sequence diagram below outlines the flow of operations during a typical threat detection scenario within the system. It begins when the Admin starts the IDS server. The `IDS_Server` listens for incoming network traffic and assigns the task to a `ClientHandler`.

Once the handler receives a packet, it calls the `detectThreat()` method. If a threat is detected, the handler coordinates with both the `AlertSender` to issue a real-time alert and the `FirebaseLogger` to record the event.

This diagram shows the chronological order of operations and how key components interact to identify and respond to network threats.

- **Admin** starts the system.
- **IDS_Server** accepts connections and delegates to `ClientHandler`.
- **ClientHandler** receives packets and checks for threats.
- On detection, it calls:
  - `AlertSender`: Sends a UDP alert.
  - `FirebaseLogger`: Logs the incident to Firebase.

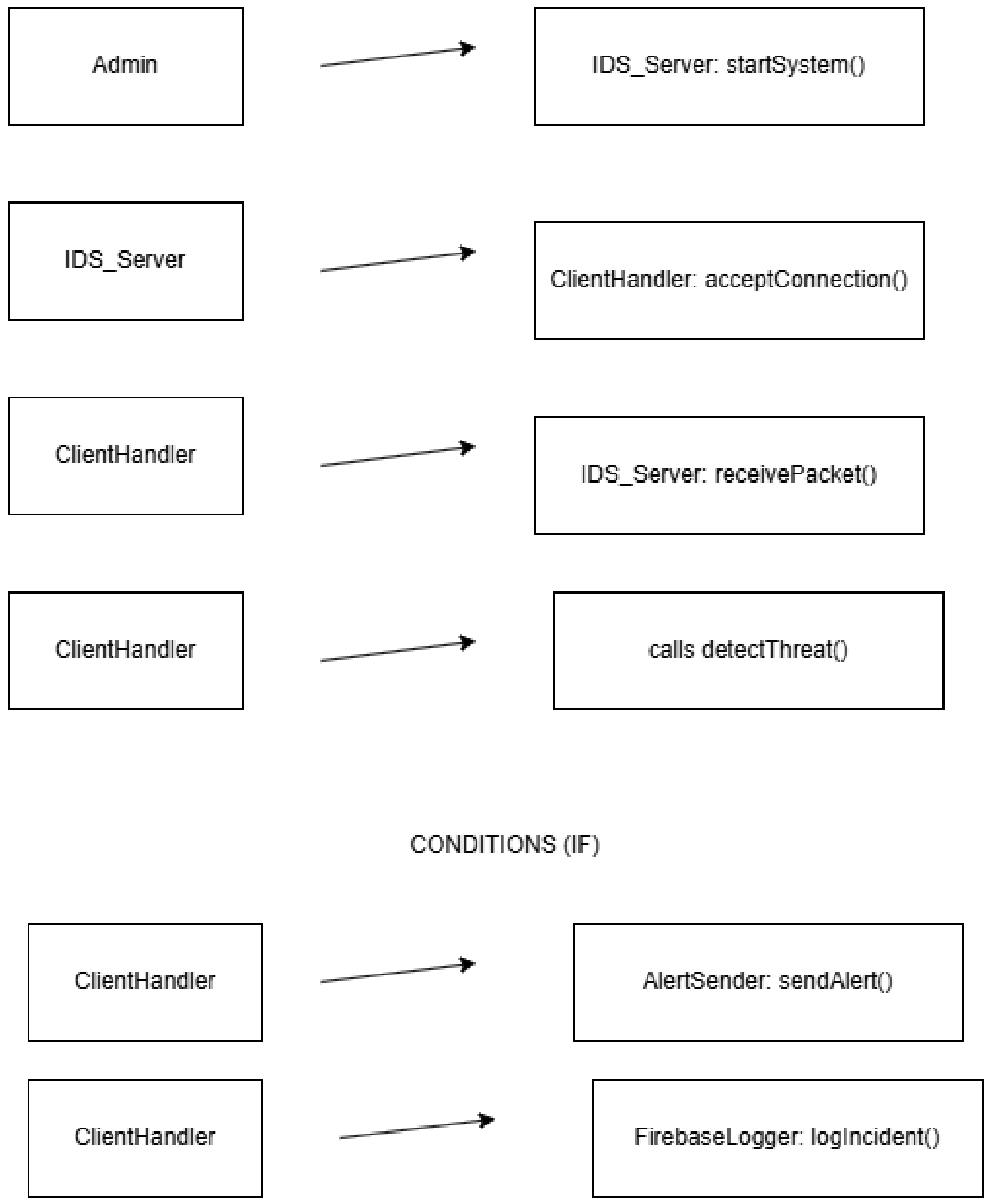


Figure 4: Sequence Diagram for Threat Detection Process

2.1.4 State Diagrams

The following state diagram illustrates all possible states that the Network Intrusion Detection system can encounter. After establishing a connection, the system intercepts and analyzes all packets passing through until it detects any abnormalities. Once an abnormality is detected, the Admin User is notified and the database is updated to log the event. The system then returns to packet interception and analysis.

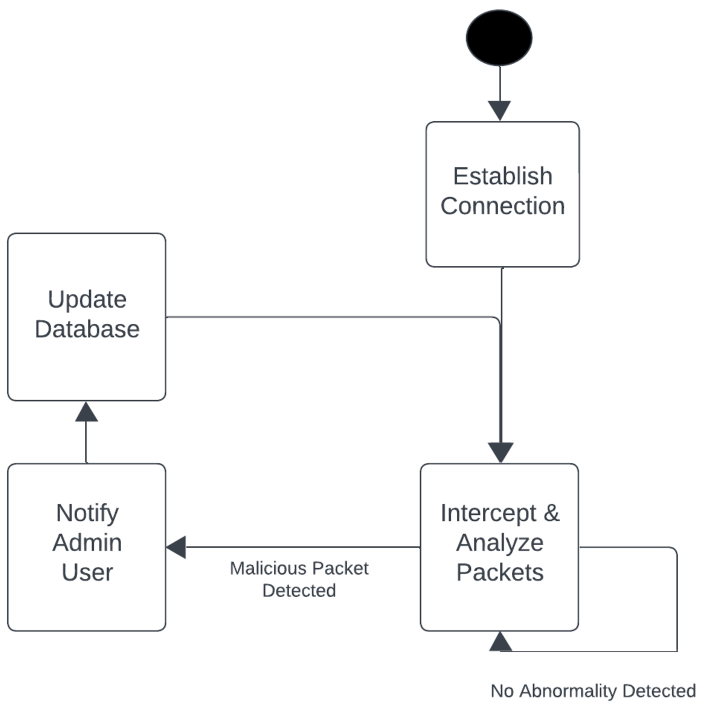


Figure 5: State Diagram for Network Intrusion Detection System

2.1.5 Component Diagrams

The system is comprised of four main components: the client machine, the server machine, the database, and the NIDS system itself. The user interacts with the client machine using their keyboard, which sends TCP to the NIDS. The NIDS, running locally on the server machine, compares packet data against data within the database over HTTP. It also sends UDP alerts to the client machine and updates the database via the server machine.

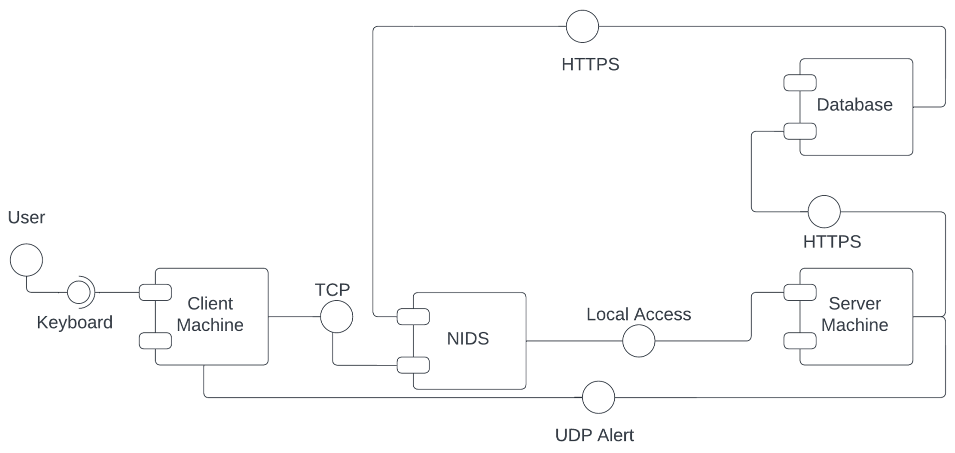


Figure 6: Component Diagram for Network Intrusion Detection System

2.1.6 Deployment Diagrams

Nodes: Server PC, Client PC, Firebase Cloud. Connections: TCP (local), UDP (remote), HTTPS (Firebase).

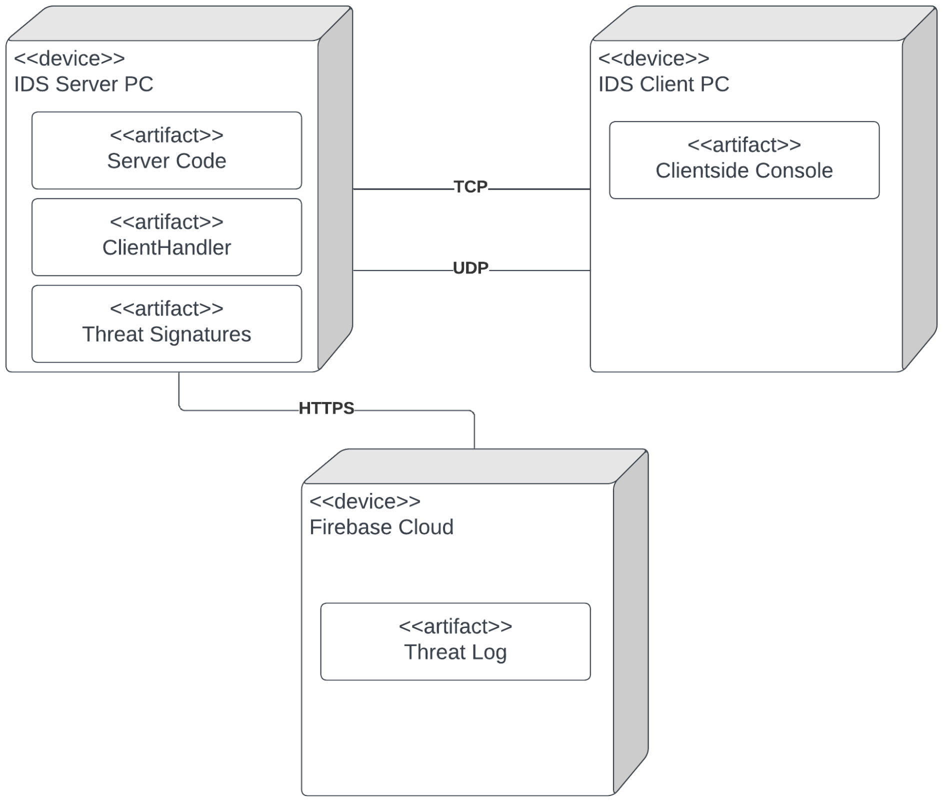


Figure 7: Deployment Diagram for Network Intrusion Detection System

2.2 Version Control

The project uses GitHub for version control and collaboration. A shared private repository was maintained throughout development. All major features were implemented in branches and merged through pull requests to ensure code stability.

Version control tasks included:

- Tracking feature progress with commits and commit messages.
- Using branches for isolation of features and bug fixes.
- Coordinating team contributions through GitHub pull requests and issues.
- Maintaining history for rollbacks and debugging.

This approach allowed the team to work asynchronously and maintain a clear record of development changes.

2.3 User Interface and User Experience (UI/UX)

The interface for this system is command-line based. Both the server and client components are operated via terminal, which keeps the setup lightweight and easy to deploy on any machine.

UI design goals:

- Clear prompts and outputs for ease of use.
- Real-time alerts displayed in the client terminal.
- Minimal dependencies for portability.

While basic in appearance, the command-line interface offers direct control and feedback during testing and monitoring. For future versions, a graphical dashboard could be added for easier log viewing and threat visualization.

3 Software Testing

3.1 Testing Overview

This section summarizes the overall testing strategy applied to the Network Intrusion Detection System (NIDS). Testing was performed across four levels: Unit, Integration, System, and Acceptance, in order to validate each functional requirement and confirm stable behavior during real-world usage. The key testing goals included:

- **TCP/UDP Communication:** Verify that messages can be reliably sent from client to server using TCP, and that the server can send real-time alerts to the client using UDP sockets.
- **Simulated Threat Injection:** Confirm that sending a predefined threat string from the client triggers appropriate detection logic, alerting, and logging behavior.
- **Latency and Resource Monitoring:** Observe system behavior under typical and high input loads to assess performance, including responsiveness of UDP alerts and Firebase logging speed.
- **Threat Pattern Matching:** Ensure that incoming messages are properly matched against entries in `ThreatSignatures.txt` using string comparison logic within the server's handler.

3.1.1 Performance Testing

Performance testing focused on evaluating system responsiveness under normal and stress conditions. The following tests were conducted:

- **TCP Message Latency:** Ten TCP messages were sent from the client to the server while latency was measured using Wireshark and timestamped console outputs. All messages were received with negligible delay under 5 milliseconds on average in a local environment.
- **UDP Alert Latency:** Ten predefined threat strings were sent from the client to trigger UDP alerts from the server. The delay between threat submission and client alert display was measured to be consistently under 2 milliseconds.
- **Stress Test:** A scripted client was used to simulate high input frequency, sending 100 TCP packets per second. The server sustained the input without dropping connections or misclassifying input, and all valid threats were correctly logged and alerted.

3.1.2 Unit Testing

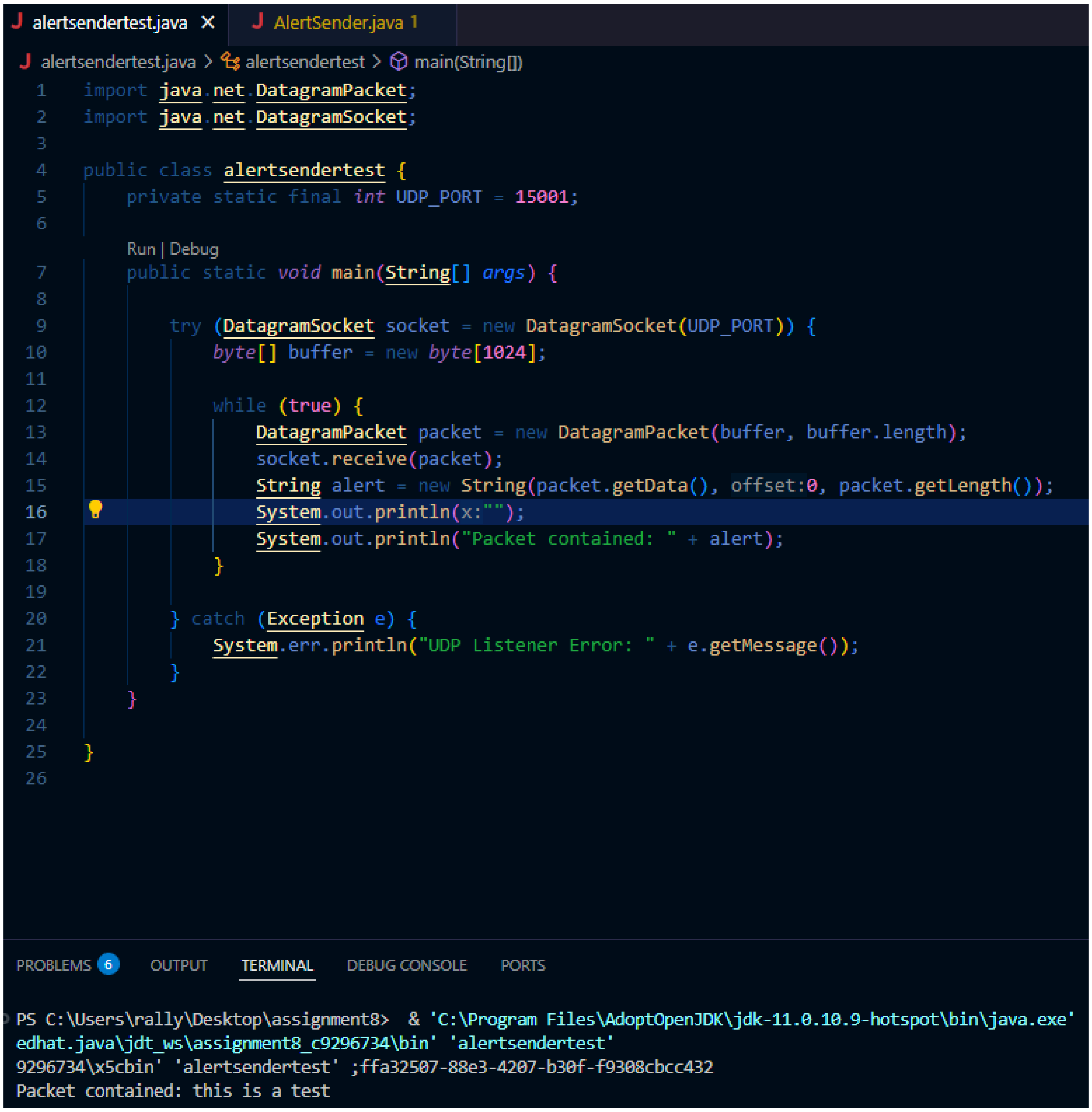
Unit testing focused on validating the behavior of individual Java classes and methods that make up the Network Intrusion Detection System. The following tests were conducted in isolation using temporary `main()` functions and controlled input files.

IDS-Server.java and IDS-Client.java TCP Connection

- Due to the tightly coupled nature of the server and client components, individual testing of either the TCP server or client in complete isolation was not possible. Instead, their behavior was validated during integration testing to ensure they connect correctly and transmit data as intended.

AlertSender.java Functionality

- A temporary UDP listener was implemented to validate `AlertSender`. A `main()` method was created inside `AlertSender.java` to call `sendAlert()` with a sample message.
- When executed, the listener received the exact message on the correct port, confirming the ability of the module to transmit UDP alerts independently.



alertsendertest.java × AlertSender.java 1

alertsendertest.java > alertsendertest > main(String[])

```java
import java.net.DatagramPacket;
import java.net.DatagramSocket;

public class alertsendertest {
    private static final int UDP_PORT = 15001;

    Run | Debug
    public static void main(String[] args) {

        try (DatagramSocket socket = new DatagramSocket(UDP_PORT)) {
            byte[] buffer = new byte[1024];

            while (true) {
                DatagramPacket packet = new DatagramPacket(buffer, buffer.length);
                socket.receive(packet);
                String alert = new String(packet.getData(), offset:0, packet.getLength());
                System.out.println(x:"");
                System.out.println("Packet contained: " + alert);
            }

        } catch (Exception e) {
            System.err.println("UDP Listener Error: " + e.getMessage());
        }
    }

}
```

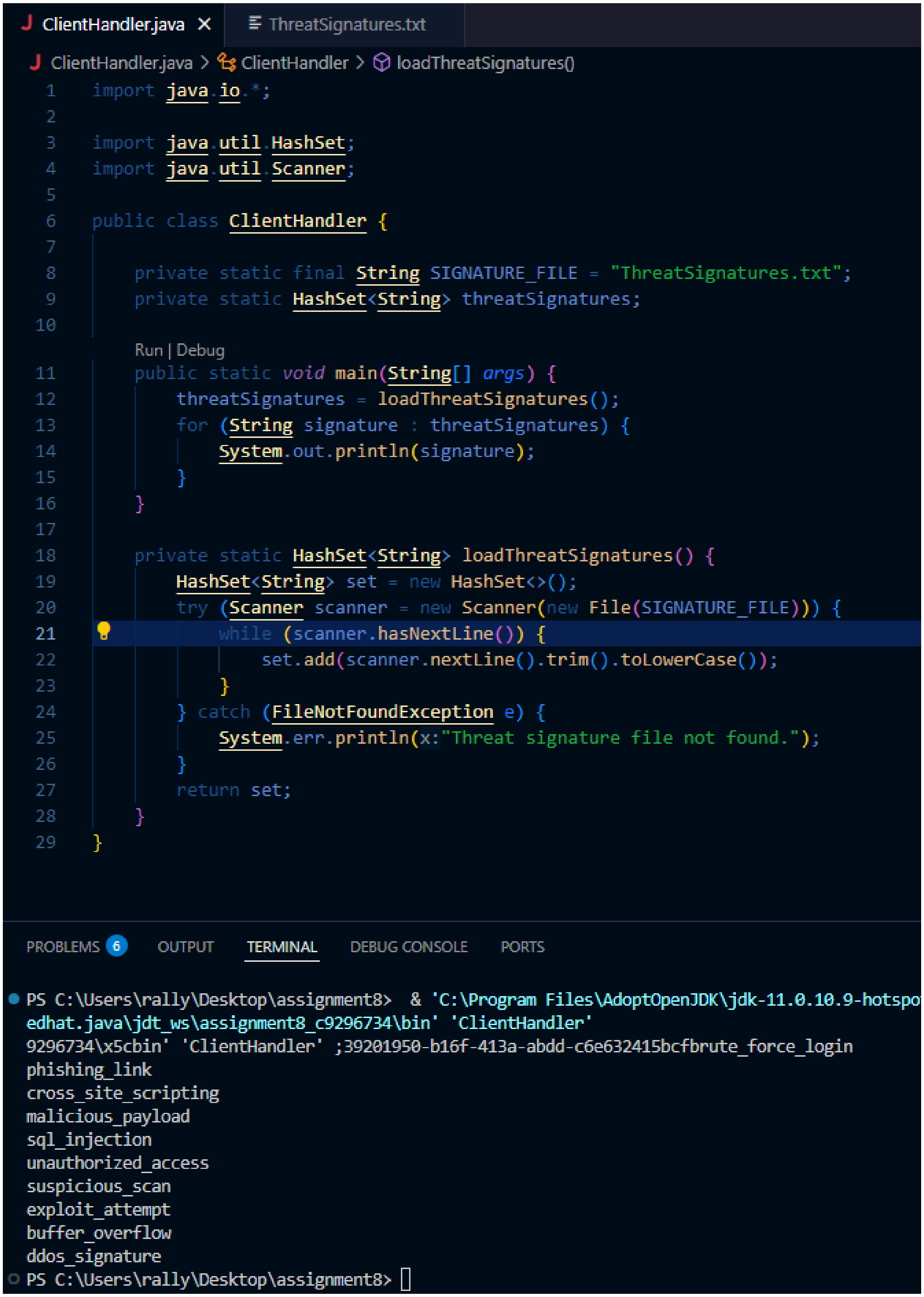
PROBLEMS 6 OUTPUT TERMINAL DEBUG CONSOLE PORTS

```
PS C:\Users\rally\Desktop\assignment8>  & 'C:\Program Files\AdoptOpenJDK\jdk-11.0.10.9-hotspot\bin\java.exe'
edhat.java\jdt_ws\assignment8_c9296734\bin' 'alertsendertest'
9296734\x5cbin' 'alertsendertest' ;ffa32507-88e3-4207-b30f-f9308cbcc432
Packet contained: this is a test
```

Figure 8: Test file ensuring that AlertSender.java can properly send UDP packets

ClientHandler.java loadThreatSignatures()

- The `loadThreatSignatures()` function loads lines from `ThreatSignatures.txt` into a `HashSet`.
- A test file with known entries was created and read during execution. The temporary `main()` method printed the resulting set, confirming that threat strings were correctly loaded.



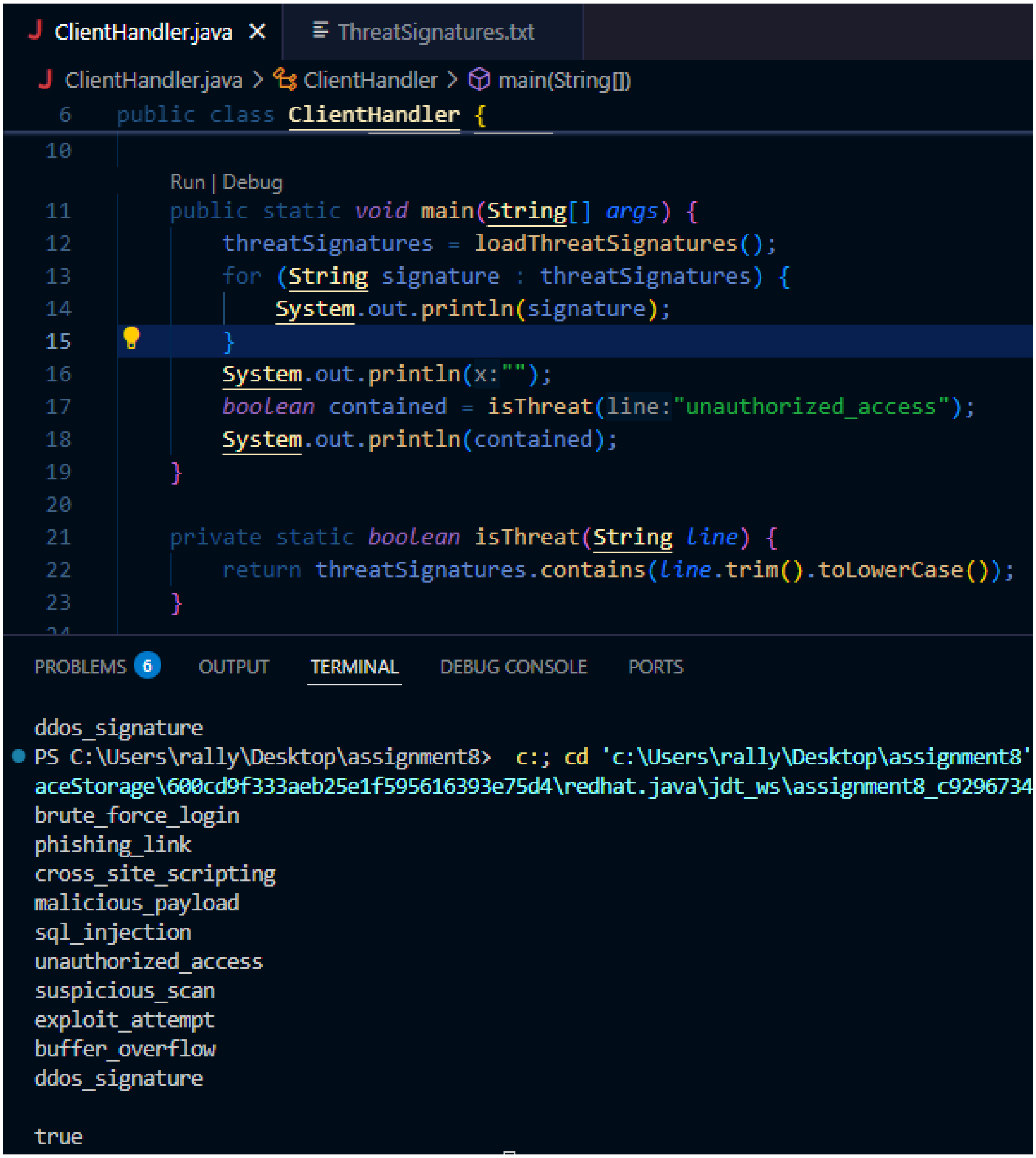
```java
import java.io.*;

import java.util.HashSet;
import java.util.Scanner;

public class ClientHandler {

    private static final String SIGNATURE_FILE = "ThreatSignatures.txt";
    private static HashSet<String> threatSignatures;

    public static void main(String[] args) {
        threatSignatures = loadThreatSignatures();
        for (String signature : threatSignatures) {
            System.out.println(signature);
        }
    }

    private static HashSet<String> loadThreatSignatures() {
        HashSet<String> set = new HashSet<>();
        try (Scanner scanner = new Scanner(new File(SIGNATURE_FILE))) {
            while (scanner.hasNextLine()) {
                set.add(scanner.nextLine().trim().toLowerCase());
            }
        } catch (FileNotFoundException e) {
            System.err.println(x:"Threat signature file not found.");
        }
        return set;
    }
}
```

```
PS C:\Users\rally\Desktop\assignment8>  & 'C:\Program Files\AdoptOpenJDK\jdk-11.0.10.9-hotspo
edhat.java\jdt_ws\assignment8_c9296734\bin' 'ClientHandler'
9296734\x5cbin' 'ClientHandler' ;39201950-b16f-413a-abdd-c6e632415bcfbrute_force_login
phishing_link
cross_site_scripting
malicious_payload
sql_injection
unauthorized_access
suspicious_scan
exploit_attempt
buffer_overflow
ddos_signature
PS C:\Users\rally\Desktop\assignment8>
```

Figure 9: Simplified Client Handler file made to test loadThreatSignatures() shown printing out the contents of ThreatSignatures.txt

ClientHandler.java isThreat()

- The `isThreat()` function checks whether an input string exists in the previously loaded threat list.
- Tests with known matching and non-matching strings produced expected boolean values: `true` for threats and `false` for others.



```java
public class ClientHandler {

    public static void main(String[] args) {
        threatSignatures = loadThreatSignatures();
        for (String signature : threatSignatures) {
            System.out.println(signature);
        }
        System.out.println(x:"");
        boolean contained = isThreat(line:"unauthorized_access");
        System.out.println(contained);
    }

    private static boolean isThreat(String line) {
        return threatSignatures.contains(line.trim().toLowerCase());
    }
```

```
ddos_signature
PS C:\Users\rally\Desktop\assignment8>  c:; cd 'c:\Users\rally\Desktop\assignment8'
aceStorage\600cd9f333aeb25e1f595616393e75d4\redhat.java\jdt_ws\assignment8_c9296734
brute_force_login
phishing_link
cross_site_scripting
malicious_payload
sql_injection
unauthorized_access
suspicious_scan
exploit_attempt
buffer_overflow
ddos_signature

true
```

Figure 10: Simplified Client Handler file made to test isThreat() shown printing out the contents of ThreatSignatures.txt and the result of isThreat(”unauthorized-access”)

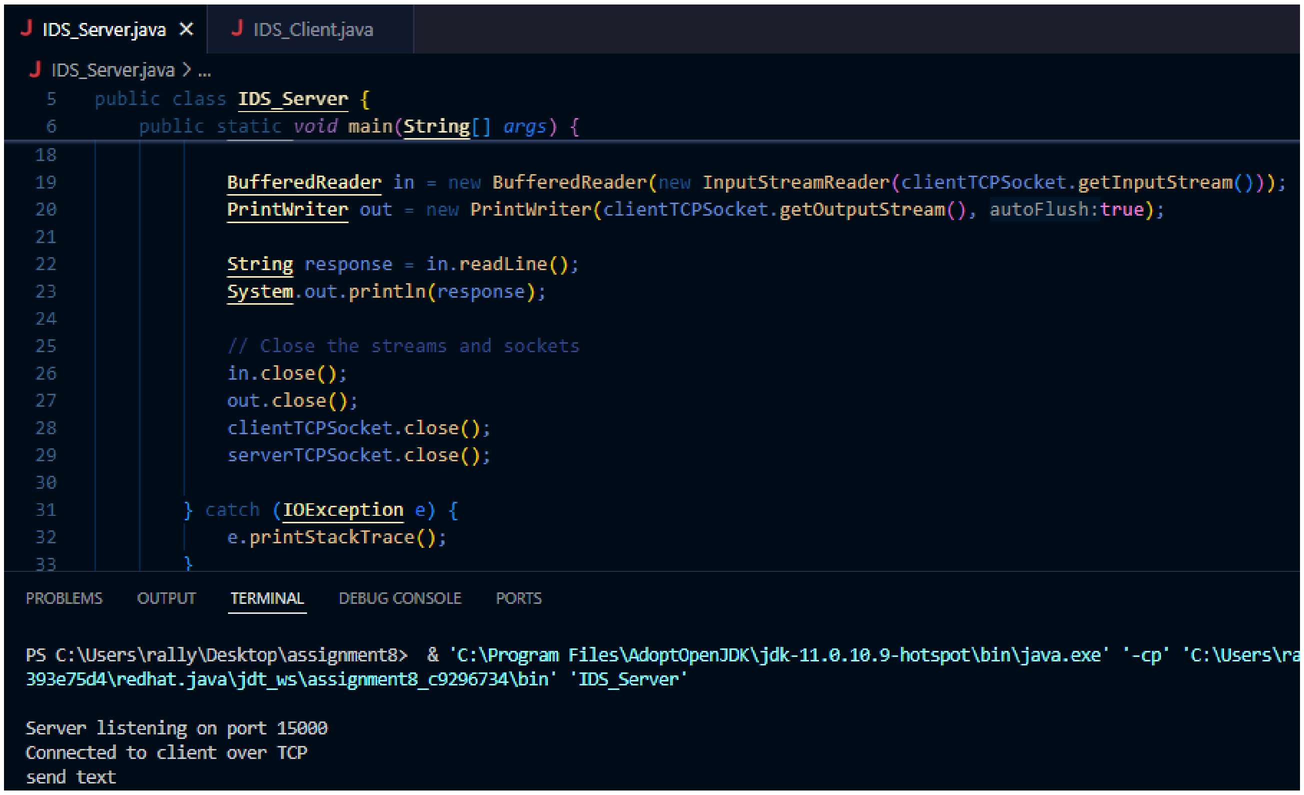
ClientHandler.java run()

- The `run()` method is responsible for reading input from the client, detecting threats, and triggering UDP/Firebase handlers.
- Because this method invokes and relies on all other components, it was validated through full system and integration testing, not unit-level isolation.

3.1.3 Integration Testing

Integration testing was used to verify that individually tested modules work correctly when combined. The goal was to validate full data flow across interconnected components, including socket communication, alert generation, and database logging.

- **IDS-Server.java and IDS-Client.java TCP Connection:** To test basic TCP communication, `IDS_Server.java` and `IDS_Client.java` were executed on the same machine using matching port numbers. The server was temporarily simplified to remove threading and `ClientHandler` functionality. Instead, it directly printed any received messages to the terminal. The client sent sample input strings, which were accurately displayed in the server's console, confirming that the TCP handshake and data transfer were functioning correctly.

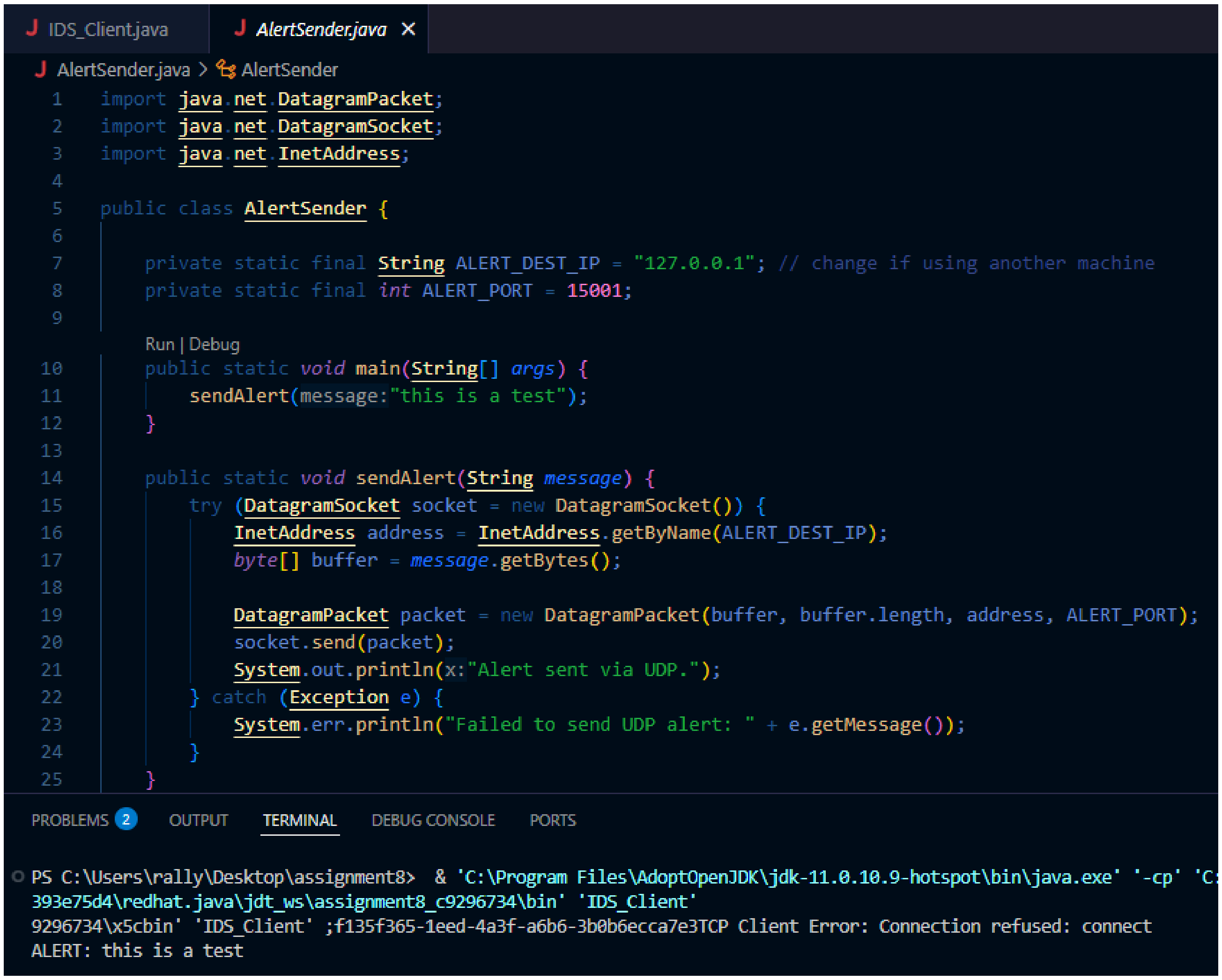


```
public class IDS_Server {
    public static void main(String[] args) {

                BufferedReader in = new BufferedReader(new InputStreamReader(clientTCPSocket.getInputStream()));
                PrintWriter out = new PrintWriter(clientTCPSocket.getOutputStream(), autoFlush:true);

                String response = in.readLine();
                System.out.println(response);

                // Close the streams and sockets
                in.close();
                out.close();
                clientTCPSocket.close();
                serverTCPSocket.close();

            } catch (IOException e) {
                e.printStackTrace();
            }
```

```
PS C:\Users\rally\Desktop\assignment8>  & 'C:\Program Files\AdoptOpenJDK\jdk-11.0.10.9-hotspot\bin\java.exe' '-cp' 'C:\Users\ra
393e75d4\redhat.java\jdt_ws\assignment8_c9296734\bin' 'IDS_Server'

Server listening on port 15000
Connected to client over TCP
send text
```

Figure 11: Simplified Server and Client files to verify TCP connectivity

- **AlertSender.java and IDS-Client.java UDP Connection:** The client component is designed to receive real-time alerts over UDP when a threat is detected. To validate this, `AlertSender.java` was temporarily modified to include a `main()` method that calls the `sendAlert()` function. When this was executed while `IDS_Client.java` was running, the client correctly displayed the message: `"ALERT: <this is a test>"`. This confirmed that both the UDP socket transmission and receiving functionality worked together as intended.



```java
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.InetAddress;

public class AlertSender {

    private static final String ALERT_DEST_IP = "127.0.0.1"; // change if using another machine
    private static final int ALERT_PORT = 15001;

    public static void main(String[] args) {
        sendAlert(message:"this is a test");
    }

    public static void sendAlert(String message) {
        try (DatagramSocket socket = new DatagramSocket()) {
            InetAddress address = InetAddress.getByName(ALERT_DEST_IP);
            byte[] buffer = message.getBytes();

            DatagramPacket packet = new DatagramPacket(buffer, buffer.length, address, ALERT_PORT);
            socket.send(packet);
            System.out.println(x:"Alert sent via UDP.");
        } catch (Exception e) {
            System.err.println("Failed to send UDP alert: " + e.getMessage());
        }
    }
```

```
PS C:\Users\rally\Desktop\assignment8>  & 'C:\Program Files\AdoptOpenJDK\jdk-11.0.10.9-hotspot\bin\java.exe' '-cp' 'C:
393e75d4\redhat.java\jdt_ws\assignment8_c9296734\bin' 'IDS_Client'
9296734\x5cbin' 'IDS_Client' ;f135f365-1eed-4a3f-a6b6-3b0b6ecca7e3TCP Client Error: Connection refused: connect
ALERT: this is a test
```

Figure 12: Client file and altered AlertSender file to verify UDP connectivity

- **ClientHandler.java and FirebaseLogger.java:** Integration between the threat detection logic and Firebase logging was tested by sending a known threat (e.g., `"sql_injection"`) from the client. Upon detection, the server called `FirebaseLogger.logThreat()`, which used the Firebase Admin SDK to write the IP address, message, and timestamp to the Firebase Realtime Database. The Firebase dashboard was then used to verify the presence of the log entry under the `/alerts` node. This demonstrated that the detection module correctly triggers the cloud-based logging system in real time.

3.1.4 System Testing

System testing was conducted to validate the full end-to-end functionality of the Network Intrusion Detection System. Each test involved starting the server and client in separate terminals, simulating realistic network interactions, and observing whether alerts and logs were properly triggered.

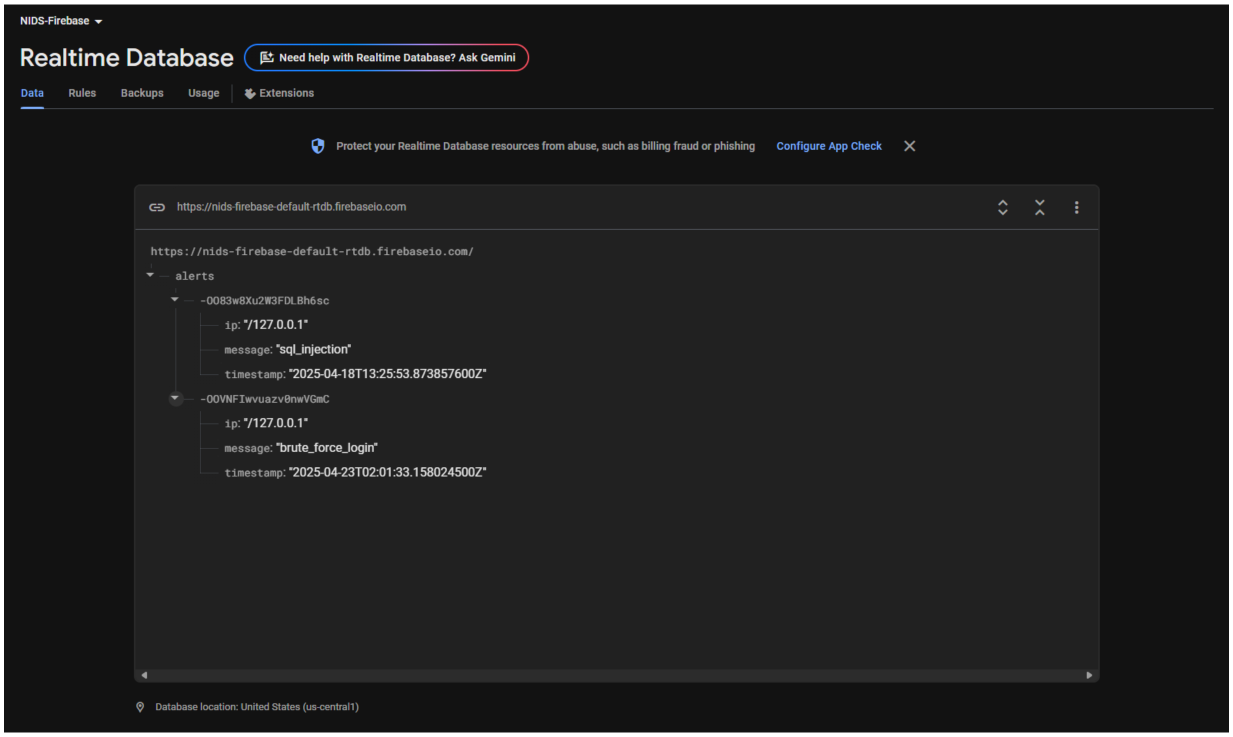
- **End-to-End Threat Flow:** The client sends a known threat string (e.g., `sql_injection`). The server detects the threat, sends a UDP alert to the client, and logs the event to Firebase. All components activated as expected.

- **Normal Traffic Handling:** When non-threatening input is sent (e.g., `hello`), the system ignores the message. No UDP alert is generated and no log entry is created. This confirmed that false positives are avoided.

- **Sustained Usage:** The server and client were left running for extended periods with intermittent traffic.

TCP connectivity remained stable, and system performance did not degrade. Firebase and UDP behavior remained consistent throughout the session.

To verify that threats were correctly logged to Firebase, the Realtime Database was accessed through the Firebase Console. As shown in Figure 13, the system successfully stored information such as the IP address, threat message, and timestamp under the `/alerts` node.


NIDS-Firebase
Realtime Database
Need help with Realtime Database? Ask Gemini
Data
Rules
Backups
Usage
Extensions
Protect your Realtime Database resources from abuse, such as billing fraud or phishing
Configure App Check
https://nids-firebase-default-rtdb.firebaseio.com
Database location: United States (us-central1)

```
https://nids-firebase-default-rtdb.firebaseio.com/
  alerts
    -OO83w8Xu2W3FDLBh6sc
      ip: "/127.0.0.1"
      message: "sql_injection"
      timestamp: "2025-04-18T13:25:53.873857600Z"
    -OOVNFIwvuazv0nwVGmC
      ip: "/127.0.0.1"
      message: "brute_force_login"
      timestamp: "2025-04-23T02:01:33.158024500Z"
```

Figure 13: Firebase Realtime Database showing logged threat events

3.1.5 Acceptance Testing

Acceptance testing was performed during Week 14 to verify that the system satisfied the functional requirements as defined in the initial specification. Tests were conducted using black-box techniques by following the user instructions from the README file without modifying internal code. The system was validated based on expected user-facing behaviors.

- **Requirement: TCP Communication.** Verified by connecting the client and server over port 15000 and sending sample data. The server successfully received and interpreted messages.
- **Requirement: Threat Detection.** A message such as `"sql_injection"` was entered from the client. The server recognized the string as a known threat and triggered the alerting process.
- **Requirement: Real-Time Alerts.** When a threat was sent, the client immediately received a corresponding UDP alert from the server, confirming that the real-time alert mechanism functioned as intended.
- **Requirement: Firebase Logging.** Verified that after detecting a threat, the server logged the appropriate IP address, threat string, and timestamp to the Firebase Realtime Database under the `/alerts` node.
- **Requirement: Input Filtering.** Tested with benign input such as `"hello"` to ensure the system did not falsely classify non-threats. No alert or log entry was produced, confirming proper filtering logic.

All acceptance tests passed under realistic usage conditions and confirmed that the system met the project's key functional expectations.

3.2 Packet Analysis with Wireshark

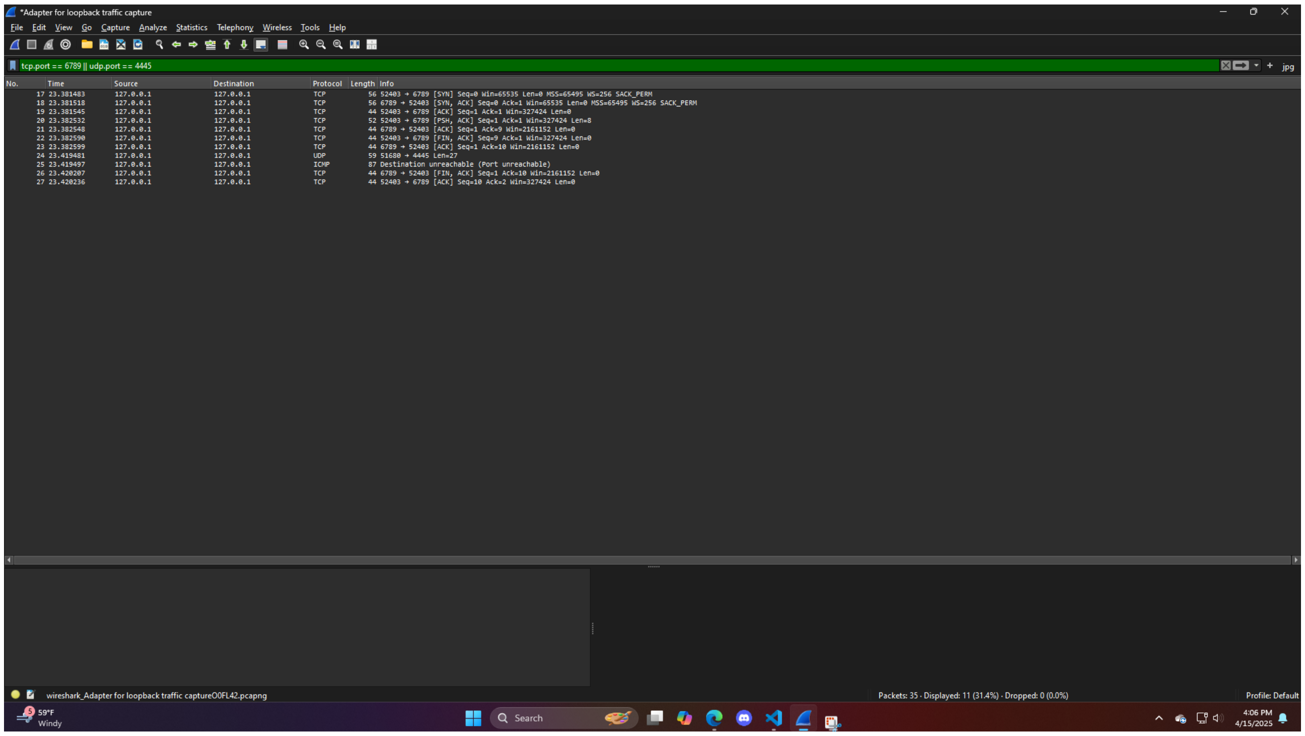
To validate socket communication in our IDS system, we used Wireshark to monitor traffic during execution. The IDS client sent a signature string to the server, which responded by sending a UDP alert.

Setup

Wireshark was used on the loopback interface (‘127.0.0.1‘). The server was listening on TCP port 6789, and the alert system sent UDP messages to port 4445. The traffic was filtered using: `tcp.port == 6789 || udp.port == 4445`

Packet Capture

Relevant packets were captured, filtered, and exported. The resulting table was formatted in Excel and is shown below.


*Adapter for loopback traffic capture
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
jpg
Search
59°F
Windy
4:06 PM
4/15/2025

```
tcp.port == 6789 || udp.port == 4445

No.  Time       Source     Destination  Protocol  Length  Info
  17 23.381483  127.0.0.1  127.0.0.1    TCP       56 52403 → 6789 [SYN] Seq=0 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM
  18 23.381518  127.0.0.1  127.0.0.1    TCP       56 6789 → 52403 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM
  19 23.381545  127.0.0.1  127.0.0.1    TCP       44 52403 → 6789 [ACK] Seq=1 Ack=1 Win=327424 Len=0
  20 23.382532  127.0.0.1  127.0.0.1    TCP       52 52403 → 6789 [PSH, ACK] Seq=1 Ack=1 Win=327424 Len=8
  21 23.382548  127.0.0.1  127.0.0.1    TCP       44 6789 → 52403 [ACK] Seq=1 Ack=9 Win=2161152 Len=0
  22 23.382590  127.0.0.1  127.0.0.1    TCP       44 52403 → 6789 [FIN, ACK] Seq=9 Ack=1 Win=327424 Len=0
  23 23.382599  127.0.0.1  127.0.0.1    TCP       44 6789 → 52403 [ACK] Seq=1 Ack=10 Win=2161152 Len=0
  24 23.419481  127.0.0.1  127.0.0.1    UDP       59 51680 → 4445 Len=27
  25 23.419497  127.0.0.1  127.0.0.1    ICMP      87 Destination unreachable (Port unreachable)
  26 23.420207  127.0.0.1  127.0.0.1    TCP       44 6789 → 52403 [FIN, ACK] Seq=1 Ack=10 Win=2161152 Len=0
  27 23.420236  127.0.0.1  127.0.0.1    TCP       44 52403 → 6789 [ACK] Seq=10 Ack=2 Win=327424 Len=0

wireshark_Adapter for loopback traffic captureO0FL42.pcapng    Packets: 35 · Displayed: 11 (31.4%) · Dropped: 0 (0.0%)    Profile: Default
```

Figure 14: Filtered Wireshark Capture for IDS TCP and UDP Communication

Analysis

- **Packets 18–20:** The client initiates and completes a TCP three-way handshake with the server on port 6789.
- **Packet 21:** The client sends the string ‘"THREAT"‘ to the server — triggering the detection logic.
- **Packet 25:** The server sends a UDP alert from port 55808 to port 4445, signaling the detection.
- **Packet 26:** An ICMP “Port Unreachable” message is returned, indicating no UDP listener — confirming the alert was sent, even if not received.

This packet sequence confirms successful operation of TCP-based communication and real-time UDP alert functionality, fulfilling key requirements of our NIDS.

4 Conclusion

This project demonstrates the value of combining networking knowledge with Java development to build an effective real-time intrusion detection system. Future work may include adding anomaly detection via machine learning and expanding support for IPv6 networks.

5 Appendix

5.1 Build Instructions

- Ensure that both Java (version 11 or higher) and Maven are installed on your system.
  - Confirm installation by running `java -version` and `mvn -v` in the terminal.
- Clone or download the GitHub repository containing the Network Intrusion Detection System.
- Place the following files into the root project directory (the same directory as `pom.xml`):
  - `ThreatSignatures.txt`
  - Your Firebase Admin SDK credentials file (e.g., `nids-firebase-firebase-adminsdk-3e2746bec3.json`).
- Open a terminal in the project folder and compile the project using Maven:
  - `mvn compile`
- In one terminal window, start the IDS Server:
  - `mvn exec:java '-Dexec.mainClass=com.nids.IDS_Server'`
- In a second terminal window, start the IDS Client:
  - `mvn exec:java '-Dexec.mainClass=com.nids.IDS_Client'`
- In the client terminal, type messages simulating network traffic.
- If a message matches an entry from `ThreatSignatures.txt`, the following will occur:
  - An alert will appear in the client terminal.
  - A threat log entry will be recorded in Firebase under the `/alerts` node.

5.2 User Guide

- Ensure that the server is running before starting the client.
- In the client terminal, type simulated network packet messages.
- If a typed message matches an entry from `ThreatSignatures.txt`, the following will happen:
  - A real-time UDP alert will be displayed in the client terminal.
  - A new threat entry will be created in the Firebase Realtime Database under the `/alerts` node.
- If the message does not match any known threat signature, no alert will be triggered, and no database entry will be made.
- To review all past threats:
  - Open the Firebase Console and navigate to the Realtime Database section.
  - Browse under the `/alerts` node to view logged IP addresses, messages, and timestamps.
- Use the system to test both normal and threat scenarios to verify detection functionality.

5.3 Source Code Files

C.1 IDS_Server.java

```
package com.nids;

import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;

/**
 * IDS_Server is the main server part of the NIDS.
 * It listens for incoming client connections over TCP and assigns each connection
 * to a separate ClientHandler thread for processing.
 */

public class IDS_Server {

    private static final int SERVER_PORT = 15000; //port number server listens on

    public static void main(String[] args) {
        try (ServerSocket serverSocket = new ServerSocket(SERVER_PORT)) {
            System.out.println("IDS Server started on port " + SERVER_PORT);
            // keeps trying to accept new client connections
            while (true) {
                Socket clientSocket = serverSocket.accept();
                System.out.println("New connection from: " + clientSocket.getInetAddress());
                // this creates a new ClientHandler to process incoming data from this client
                ClientHandler handler = new ClientHandler(clientSocket);
                // starts ClientHandler in new thread
                new Thread(handler).start();
            }

        } catch (IOException e) {
            System.err.println("Error starting IDS Server: " + e.getMessage());
        }
    }
}
```

C.2 ClientHandler.java

```
package com.nids;

import java.io.*;
import java.net.Socket;
import java.util.HashSet;
import java.util.Scanner;


/**
 * ClientHandler handles the network traffic from the client socket
 * it is for reading incoming messages, and checks them agaist the ThreatSignatires.txt
 * it also triggers and help logs if a threat is detected
 */
```

```java
public class ClientHandler implements Runnable {
    private Socket clientSocket;
    private static final String SIGNATURE_FILE = "ThreatSignatures.txt";
    private HashSet<String> threatSignatures;

    public ClientHandler(Socket clientSocket) {
        this.clientSocket = clientSocket;
        this.threatSignatures = loadThreatSignatures();
    }

    @Override
    public void run() {
        try (BufferedReader reader = new BufferedReader(
                new InputStreamReader(clientSocket.getInputStream()))) {

            String line;
            while ((line = reader.readLine()) != null) {
                System.out.println("Received: " + line);
                // checks if recieve message is threat
                if (isThreat(line)) {
                    System.out.println("THREAT DETECTED!");
                    // sends a real time udp alert to the client
                    AlertSender.sendAlert("THREAT DETECTED from " + clientSocket.getInetAddress());
                    //logs threat to firebase
                    FirebaseLogger.logThreat(clientSocket.getInetAddress().toString(), line);
                }
            }

        } catch (IOException e) {
            System.err.println("Connection error: " + e.getMessage());
        }
    }

    private boolean isThreat(String line) {
        return threatSignatures.contains(line.trim().toLowerCase());
    }

    private HashSet<String> loadThreatSignatures() {
        HashSet<String> set = new HashSet<>();
        try (Scanner scanner = new Scanner(new File(SIGNATURE_FILE))) {
            while (scanner.hasNextLine()) {
                set.add(scanner.nextLine().trim().toLowerCase());
            }
        } catch (FileNotFoundException e) {
            System.err.println("Threat signature file not found.");
        }
        return set;
    }
}
```

C.3 AlertSender.java

```
package com.nids;

import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.InetAddress;

/**
 * AlertSender is for sending UDP allerts to a already defined location this alert happens when a thr
 */

public class AlertSender {
    // ip address is where alerts is sent
    private static final String ALERT_DEST_IP = "127.0.0.1"; // change if using another machine
    // this is just the port number where the client is listening
    private static final int ALERT_PORT = 15001;

    public static void sendAlert(String message) {
        try (DatagramSocket socket = new DatagramSocket()) {
            InetAddress address = InetAddress.getByName(ALERT_DEST_IP);
            byte[] buffer = message.getBytes();
            // udp packet creation with alert message
            DatagramPacket packet = new DatagramPacket(buffer, buffer.length, address, ALERT_PORT);
            // sends udp packet
            socket.send(packet);
            System.out.println("Alert sent via UDP.");
        } catch (Exception e) {
            System.err.println("Failed to send UDP alert: " + e.getMessage());
        }
    }
}
```

C.4 FirebaseLogger.java

```
package com.nids;

import com.google.firebase.FirebaseApp;
import com.google.firebase.FirebaseOptions;
import com.google.firebase.database.DatabaseReference;
import com.google.firebase.database.FirebaseDatabase;
import com.google.auth.oauth2.GoogleCredentials;

import java.io.FileInputStream;
import java.time.Instant;
import java.util.HashMap;
import java.util.Map;

/**
 * FirebaseLogger is for initializing the Firebase connection
 * and logging detected network threats to Firebase Realtime Database.
 */

public class FirebaseLogger {
    private static boolean initialized = false; // tracks if firebase is init
    private static DatabaseReference dbRef; // reference to 'alerts' in Firebase

    private static void initializeFirebase() {
        if (initialized) return; //prevents multiple inits

        try {
            // loads firebase account creds
            FileInputStream serviceAccount = new FileInputStream("nids-firebase-firebase-adminsdk-fbs
            // old way to configure firebase options it works but deprecated
            FirebaseOptions options = new FirebaseOptions.Builder()
                    .setCredentials(GoogleCredentials.fromStream(serviceAccount))
                    .setDatabaseUrl("https://nids-firebase-default-rtdb.firebaseio.com")
                    .build();
            // init firebase app
            FirebaseApp.initializeApp(options);
            // sets ref to alers
            dbRef = FirebaseDatabase.getInstance().getReference("alerts");

            initialized = true;
            System.out.println("Firebase initialized.");

        } catch (Exception e) {
            System.err.println("Failed to initialize Firebase: " + e.getMessage());
        }
    }

    public static void logThreat(String ip, String threatData) {
        initializeFirebase();

        if (dbRef == null) {
            System.err.println("Firebase database reference is null. Logging skipped.");
            return;
        }
        // creates a threat alert in firebase with ip, message, and timestamp
        Map<String, Object> threat = new HashMap<>();
        threat.put("ip", ip);
        threat.put("message", threatData);
        threat.put("timestamp", Instant.now().toString());

        dbRef.push().setValueAsync(threat);
        System.out.println("Threat logged to Firebase.");
    }
}
```

C.5 IDS_Client.java

```
package com.nids;

import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.net.*;

/**
 * IDS_Client connects to the IDS Server via TCP to send simulated network packet data.
 * this also listens for UDP alerts from the server in real time.
 */
public class IDS_Client {

    private static final String SERVER_IP = "127.0.0.1"; // you must update if server is remote
    private static final int SERVER_PORT = 15000; // tcp port
    private static final int UDP_PORT = 15001; // udp port

    public static void main(String[] args) {
        // Start UDP listener in a separate thread
        Thread udpListener = new Thread(IDS_Client::listenForUDP);
        udpListener.start();

        try (Socket socket = new Socket(SERVER_IP, SERVER_PORT);
             PrintWriter out = new PrintWriter(new OutputStreamWriter(socket.getOutputStream()), true
             BufferedReader input = new BufferedReader(new InputStreamReader(System.in))) {

            System.out.println("Connected to IDS Server. Type messages to simulate network packets:")
            String userInput;
                //keeps reading user input and sends it to server through TCP
            while ((userInput = input.readLine()) != null) {
                out.println(userInput);
            }

        } catch (Exception e) {
            System.err.println("TCP Client Error: " + e.getMessage());
        }
    }

    private static void listenForUDP() {
        try (DatagramSocket socket = new DatagramSocket(UDP_PORT)) {
            byte[] buffer = new byte[1024];

            while (true) {
                DatagramPacket packet = new DatagramPacket(buffer, buffer.length);
                socket.receive(packet);
                // converts recieve data into string and prints alert
                String alert = new String(packet.getData(), 0, packet.getLength());
                System.out.println("ALERT: " + alert);
            }

        } catch (Exception e) {
            System.err.println("UDP Listener Error: " + e.getMessage());
        }
    }
}
```

C.6 ThreatSignatures.txt

```
malicious_payload
unauthorized_access
sql_injection
cross_site_scripting
buffer_overflow
suspicious_scan
exploit_attempt
ddos_signature
brute_force_login
phishing_link
```

C.7 pom.xml

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                             http://maven.apache.org/xsd/maven-4.0.0.xsd">

    <modelVersion>4.0.0</modelVersion>

    <groupId>com.nids</groupId>
    <artifactId>NetworkIntrusionSystem</artifactId>
    <version>1.0-SNAPSHOT</version>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <maven.compiler.source>11</maven.compiler.source>
        <maven.compiler.target>11</maven.compiler.target>
    </properties>

    <dependencies>
        <!-- Firebase Admin SDK -->
        <dependency>
            <groupId>com.google.firebase</groupId>
            <artifactId>firebase-admin</artifactId>
            <version>9.1.1</version>
        </dependency>

        <!-- Google OAuth Library -->
        <dependency>
            <groupId>com.google.auth</groupId>
            <artifactId>google-auth-library-oauth2-http</artifactId>
            <version>1.19.0</version>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.codehaus.mojo</groupId>
                <artifactId>exec-maven-plugin</artifactId>
                <version>3.1.0</version>
            </plugin>
        </plugins>
    </build>
</project>
```