CHECKERS

An Integrated Checker Game with DNS Ad-Blocking and Game Summary Mailing Services

Project Report

CS 348: Computer Networks

Maddela Vaishwitha	EE22BT031
Sujata Kar	CS22BT058
Vatista Kachroo	CS22BT060
Prakriti Tripathi	CS22BT067
Tanvi Nayak	MC22BT026

April 20, 2025

Contents

1 Introduction

This project integrates a multiplayer Checkers game with DNS-based ad-blocking and a game summary mailing service. It combines socket programming for real-time gameplay, DNS filtering to block unwanted content, and SMTP to deliver post-game summaries to players. The game interface is built using Gradio, allowing for a clean and interactive user experience.

1.1 Motivation

As part of CS348: Computer Networks, we aimed to design a project that demonstrates the practical application of key networking concepts covered in the course. The core of our system is a two-player Checkers game implemented using socket programming to enable real-time interaction between the server and clients. In addition, we integrated DNS filtering and an SMTP-based mailing feature to extend the scope of the project across multiple layers of the network stack. The use of a Gradio-based interface further enhanced the usability of the application, allowing us to present our implementation as a complete, interactive system rather than a purely backend-focused solution.

1.2 Project Goals

- Develop a two-player Checkers game using socket programming for real-time communication.
- Integrate DNS-based ad-blocking to filter specific domains during gameplay.
- Implement an SMTP-based service that emails game summaries to players after the game ends.
- Use Gradio to build a visual and interactive front-end for the game.
- Apply and explore relevant Python libraries for networking, interface design, and email handling.

2 Technical Stack

Our project was primarily developed using Python, chosen for its rich set of libraries and ease of handling low-level networking operations, protocol-level integrations, and rapid UI development. Below is a breakdown of the major tools and libraries used:

- **Socket Programming** – The core of the two-player Checkers game relies on Python's built-in `socket` module for real-time server-client communication, enabling synchronized gameplay over the network.
- **DNS Ad-blocking** – We implemented a custom DNS resolver using the `dnslib` library to filter out unwanted domains. To analyze and verify DNS packet behavior and filtering performance
- **Email Service (SMTP)** – Python's `smtplib` was used to implement the game summary mailing feature, which sends summary reports to the clients after the game ends. Libraries like `email.mime` were used to compose structured mail content and attachments.

- **User Interface (UI)** – The interactive UI for the Checkers game was built using `Gradio`, a Python-based interface library that helped us create a visually intuitive and accessible frontend directly connected to the backend logic.
- **System Utilities and Networking Support** – We used various other standard libraries such as:
  - `threading` – for concurrent handling of game states and DNS requests.
  - `argparse, os, sys, datetime, traceback, random, re` – for argument parsing, system-level interactions, exception tracking, and utility functions.
  - `requests` – for making external HTTP requests, where required.
  - `subprocess, pathlib, concurrent.futures` – for file handling and background task execution.

This diverse stack allowed us to bring together concepts from application, transport, and DNS layers into a cohesive and interactive network-based system.

3 System Architecture

The project comprises three core components: a two-player Checkers game, a DNS-based ad-blocker, and an email-based game summary service. The system features an interactive Gradio interface where players enter their moves and optionally provide email credentials to receive a post-game summary. Running in the background are Python-based server and DNS scripts—handling real-time gameplay communication and filtering DNS queries to identify and block ad domains. An "Adspace" on the interface visually distinguishes blocked and unblocked domains. Players can restart, quit, or play until one wins, after which a summary is mailed to opted-in users.

3.1 Checkers Game Implementation

The system implements a fully functional multiplayer checkers game utilizing socket programming for client-server communication and Gradio for creating an intuitive graphical user interface.

3.1.1 User Interface Design

The application leverages Gradio to generate a comprehensive interface with distinct functional areas:

- **Game Board Visualization**: A dynamically rendered checkers board using Matplotlib's plotting capabilities (`matplotlib.pyplot`, `matplotlib.patches`), which provides sophisticated visualization tools for rendering the board grid, pieces, and kings with custom colors and shapes.
- **Game Status Region**: Provides real-time feedback on game progression through the `update_game_status()` function, which dynamically reports waiting states, player turns, and game outcomes.
- **Player Status Panel**: Managed by the `get_player_status()` function, which tracks player connections and dynamically displays connection status for both BLACK and WHITE players.

- **Game Controls**: Input fields for move coordinates and action buttons for game management (Make Move, Restart Game, End Game).

3.1.2 Network Architecture

The system implements a classic client-server architecture using TCP sockets:

- **Server Socket**: Binds to port 65244 on localhost (127.0.0.1) and listens for incoming connections, allowing concurrent connections from up to two clients.
- **Client Handling**: Each connected client is assigned a dedicated thread via `threading.Thread(target=handle_client, args=(client, client_id), daemon=True).start()`, enabling simultaneous processing of player inputs without blocking the main server thread.
- **Message Broadcasting**: The `broadcast_to_clients()` function implements targeted message delivery, allowing for player-specific notifications by sending customized messages to each client based on their role (BLACK or WHITE).

3.1.3 Concurrency Management

The system utilizes multiple threads to maintain responsiveness across different functional components:

- **Socket Thread**: A dedicated thread (`socket_thread()`) manages incoming connections and client registration without blocking the GUI.

Listing 1: Socket Thread function

```
def socket_thread():
    global board, game_state
    server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    try:
        server.bind(('127.0.0.1', 65244))
        server.listen(2)
        print("Server listening on 127.0.0.1:65244")
        while True:
            if len(clients) < 2:
                client, addr = server.accept()
                print(f"Connected to {addr}")

                # Add client to our list
                clients.append(client)
                client_id = len(clients) - 1

                # Start a thread to handle this client
                threading.Thread(target=handle_client, args=(
                    ↪ client, client_id), daemon=True).start()
            else:
                # Wait before checking again
                time.sleep(1)
    except Exception as e:
        print(f"Server error: {str(e)}")
    finally:
        server.close()
```

- **Client Handler Threads**: Individual threads process each player's moves, ensuring responsive gameplay regardless of client connection speed.
- **UI Refresh Thread**: An autonomous thread periodically triggers interface updates through the `ui_refresh_thread()` function, which simulates refresh button clicks every two seconds to keep the display synchronized with the game state.

3.1.4 Game Flow Control

The server carefully manages game state transitions and enforces game rules:

- **Turn Management**: The system tracks the current player's turn and rejects moves from the non-active player, preventing out-of-sequence play attempts.
- **Move Validation**: The `move_piece_gui()` function validates move legality according to checkers rules before applying changes to the game state.
- **Game State Transitions**: The system manages transitions between "waiting," "playing," and "over" states, coordinating player connections, game initialization, and conclusion.
- **Game Control Actions**: Players can restart the game through the `restart_game()` function or prematurely end it via the `end_game()` function, with appropriate notifications sent to both players.

3.1.5 Client Implementation

The client application provides a console-based interface that:

- Establishes a socket connection to the server
- Uses a dedicated listener thread (`message_listener()`) to process incoming server messages
- Renders the board state in text format with intuitive piece representations
- Handles user input for moves and game control commands
- Facilitates smooth disconnection and reconnection flows

3.1.6 Protocol Design

The communication protocol between server and client utilizes plain text messages with structured formats:

- Move commands: `"E2 to E4"`
- Game control commands: `"end game"`, `"quit"`
- Board state updates and turn notifications
- Player-specific messages that differ based on player color

This text-based protocol design ensures clarity and ease of debugging while being efficient for the needs of the application.

3.2 DNS Adblocker

The DNS Adblocker is a critical component of our network infrastructure, providing efficient ad-blocking at the DNS level. Below are its technical specifications and implementation details.

3.2.1 Core Implementation

- **Programming Language**: Python 3
- **Dependencies**: `dnslib` and `requests` Python packages
- **DNS Protocol Handling**: Utilizes the `dnslib` library for DNS protocol support
- **Network Protocols**:
  - Listens on both UDP and TCP transports
  - Default port: 53 in main version, 12553 in alternate version
  - Binding address configurable (default: 0.0.0.0)
- **Custom Resolver**:
  - Extends `BaseResolver` from dnslib
  - Checks domain against allowlist and blocklist
  - Responds with 0.0.0.0 IP address for blocked domains (TTL: 60 seconds)
  - Forwards unblocked requests to upstream DNS

3.2.2 Technical Implementation

- **Blocklist Processing**:
  - Parses domains from standard hosts file format
  - Validates IP addresses using regex pattern `r'^(\d{1,3}\.){3}\d{1,3}$'`
  - Excludes 'localhost', 'localhost.localdomain', and 'local' domains
  - Normalizes domains to lowercase
- **Update Mechanism**:
  - Dedicated `update_blocklists.sh` script
  - Invokes Python script with `--download` flag
  - Handles service restart if running under systemd
  - Suggested cron job for daily updates at 4AM
- **Error Handling**:
  - Exception handling for blocklist/allowlist loading errors
  - Connection error handling for DNS forwarding
  - Per-source download error handling with continuation

3.2.3 Blocklist Sources

The implementation utilizes five predefined blocklist sources:

Listing 2: Blocklist sources in the implementation

```
BLOCKLIST_SOURCES = [
    "https://raw.githubusercontent.com/StevenBlack/hosts/master/hosts",
    "https://adaway.org/hosts.txt",
    "https://pgl.yoyo.org/adservers/serverlist.php?hostformat=hosts&showintro=0&mimetype=plaintext",
    "https://winhelp2002.mvps.org/hosts.txt",
    "https://someonewhocares.org/hosts/hosts"
]
```

3.2.4 Allowlist System

The implementation includes an allowlist mechanism that takes precedence over blocklists:

Listing 3: Domain resolution logic with allowlist

```
def resolve(self, request, handler):
    """Resolve a DNS request, first checking against blocklist"""
    domain = str(request.q.qname)
    self.total_count += 1

    # Remove trailing dot from domain
    if domain.endswith('.'):
        domain = domain[:-1]

    domain = domain.lower()

    # Check if domain is in allowlist
    if self.allowlist and domain in self.allowlist:
        # Allow this domain even if it's in blocklist
        pass
    # Check if domain is in blocklist
    elif domain in self.blocklist:
        self.blocked_count += 1
        print(f"Blocked: {domain}")

        # Create a response with 0.0.0.0 for blocked domains
        reply = request.reply()
        reply.add_answer(RR(request.q.qname, QTYPE.A, rdata=dns.A("0.0.0.0"), ttl=60))
        return reply

    # If not blocked, forward to upstream DNS
    try:
        if handler.protocol == 'udp':
            proxy_r = request.send(self.upstream_dns, 53)
        else:
            proxy_r = request.send(self.upstream_dns, 53, tcp=True)
        reply = DNSRecord.parse(proxy_r)
```

```
            return reply
        except Exception as e:
            print(f"Error forwarding: {e}")
            return request.reply()
```

3.2.5 Systemd Integration

The implementation provides a systemd service configuration:

Listing 4: Systemd service configuration

```
[Unit]
Description=DNS Ad Blocker
After=network.target

[Service]
ExecStart=/home/$USER/dns-ad-blocker/dns_adblocker.py --download --
    ↪ port 53
WorkingDirectory=/home/$USER/dns-ad-blocker
StandardOutput=journal
StandardError=journal
Restart=always
User=$USER

[Install]
WantedBy=multi-user.target
```

3.2.6 Performance Monitoring

The system includes performance tracking with statistics reported every 60 seconds:

Listing 5: Statistics reporting implementation

```
def get_stats(self):
    """Return current statistics"""
    uptime = time.time() - self.start_time
    hours, remainder = divmod(uptime, 3600)
    minutes, seconds = divmod(remainder, 60)

    return {
        "blocked": self.blocked_count,
        "total": self.total_count,
        "percent_blocked": round((self.blocked_count / max(1, self.
            ↪ total_count)) * 100, 2),
        "uptime": f"{int(hours)}h {int(minutes)}m {int(seconds)}s"
    }

def print_stats_periodically(resolver):
    """Print statistics every minute"""
    while True:
        time.sleep(60)
        stats = resolver.get_stats()
        print(f"Stats: {stats['blocked']} blocked out of {stats['total
            ↪ ']} queries ({stats['percent_blocked']}%) - Uptime: {
            ↪ stats['uptime']}")
```

3.2.7 Installation and Deployment

The implementation includes a comprehensive installation script that:

- Creates a dedicated directory structure (`/dns-ad-blocker/`)
- Installs required Python dependencies
- Generates the Python application script
- Creates blocklist update script
- Configures systemd service file
- Provides instructions for manual and service-based execution
- Sets up automatic updates via crontab

Figure 1: Example performance metrics dashboard

3.3 Game Summary Mailing Service

This project has an automated email service that delivers game summaries to players upon completion of a match. This feature leverages standard network protocols and libraries to facilitate communication between the game server and email servers.

3.3.1 SMTP Implementation

The email functionality is built using Python's `smtplib` library, which implements the Simple Mail Transfer Protocol (SMTP). This implementation establishes a connection to Gmail's SMTP server (`smtp.gmail.com`) on port 587, which is the standard port for SMTP with TLS encryption. The connection sequence follows the standard SMTP handshake:

1. Establish TCP connection to the SMTP server
2. Initiate TLS (Transport Layer Security) encryption using `starttls()`
3. Authenticate with the server using sender credentials
4. Transmit mail data
5. Close the connection

Listing 6: SMTP Implementation in `email_handler.py` file

```
class EmailHandler:
    def __init__(self, smtp_server="smtp.gmail.com", smtp_port=587):
        self.smtp_server = smtp_server
        self.smtp_port = smtp_port
        self.sender_email = "<email-id-for-the-server>"
        self.sender_password = "<app-password>"

        # Game history for summary
        self.game_history = []
        self.game_start_time = None
```

3.3.2 Authentication and Security

For authentication, the system uses an app password specifically generated for Gmail, rather than the account's primary password. This approach follows modern security practices by using application-specific credentials with limited permissions. The sender's email address and app password are stored within the `EmailHandler` class, though a production environment would benefit from environment variables or secure credential storage.

3.3.3 Player Email Management

Player email addresses are captured through an opt-in mechanism and stored in a dictionary structure that maps player colors to their corresponding email addresses:

```
self.player_emails = {
    "BLACK": None,
    "WHITE": None
}
```

This design allows for easy mapping of game outcomes to the appropriate recipients while maintaining a clean data structure.

3.3.4 File Generation and Transmission

When a game concludes, the system generates a comprehensive text file containing the complete game record. This file creation process involves:

1. Compiling game metadata (timestamps, players, outcome)
2. Formatting the move history with timestamps
3. Capturing the final board state
4. Writing this data to a uniquely named file based on the current timestamp

The MIME (Multipurpose Internet Mail Extensions) protocol is employed to construct a multipart email message. This allows for both plain text content in the email body and the attachment of the game summary file. The `email.mime` modules handle the proper encoding and formatting of these components according to RFC standards.

3.3.5 Network Transmission Process

The actual file transmission occurs through the following sequence:

1. The text file is read in binary mode
2. The content is encapsulated as a `MIMEApplication` object
3. Content-Disposition headers are added to specify the attachment name
4. The complete MIME message is serialized to string format
5. The message is transmitted via SMTP using the authenticated connection

To prevent blocking the main game thread during network operations, the email sending process executes in a separate thread. This asynchronous approach ensures that potentially slow network operations or timeouts do not impact the responsiveness of the game interface.

Listing 7: Internal method to send emails to players

```
def _send_emails(self, summary, filename):
        # Check if any players provided an email
        recipients = [email for email in self.player_emails.values()
            ↪ if email]
        if not recipients:
            print("No recipients to send emails to.")
            return
        try:
            # Connect to server
            with smtplib.SMTP(self.smtp_server, self.smtp_port) as
                ↪ server:
                server.starttls()
                server.login(self.sender_email, self.sender_password)

                # Send to each recipient separately
                for recipient in recipients:
                    # Create a new message for each recipient
                    message = MIMEMultipart()
                    message["From"] = self.sender_email
                    message["To"] = recipient
                    message["Subject"] = "Your Checkers Game Summary"

                    # Attach the body
                    body = "Thank you for playing Checkers! Please
                        ↪ find attached the summary of your game."
                    message.attach(MIMEText(body, "plain"))

                    # Attach the file
                    with open(filename, "rb") as file:
                        attachment = MIMEApplication(file.read(), Name
                            ↪ =filename)
                    attachment["Content-Disposition"] = f'attachment;
                        ↪ filename="{filename}"'
                    message.attach(attachment)

                    # Send email
                    server.sendmail(self.sender_email, recipient,
                        ↪ message.as_string())
                    print(f"Email sent to {recipient}")

            print("All emails sent successfully!")

        except Exception as e:
            print(f"Failed to send emails: {str(e)}")

        # Clean up the file
        try:
            os.remove(filename)
        except:
            pass
```

3.3.6 Trigger Mechanism

The mailing service is activated by the `on_game_end()` function, which is called when a game concludes due to a win, draw, or forfeit. This function passes the game outcome information to the email handler, which then initiates the summary generation and transmission process. The file is automatically deleted after successful transmission to maintain system cleanliness.

4 Observations and Results

Our implementation of the Checkers game with integrated DNS ad-blocking and email services demonstrated successful functionality across all components. The following observations showcase the system's capabilities.

4.1 Game Functionality

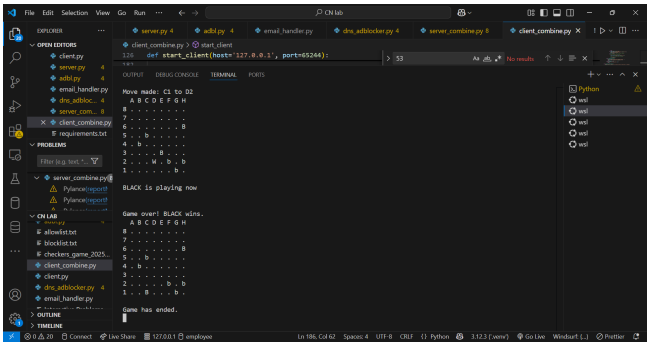

Figure 2: Terminal view from WHITE player's perspective showing the "Game Over, BLACK wins" message. This demonstrates how the game communicates the end result to the losing player.

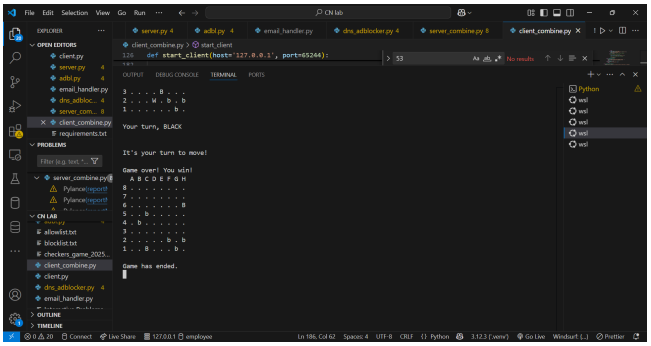

Figure 3: Terminal view from BLACK player's perspective showing the "Game Over, You Win!" message. This illustrates the different personalized messages sent to each player based on the game outcome.

4.2 Email Service Results

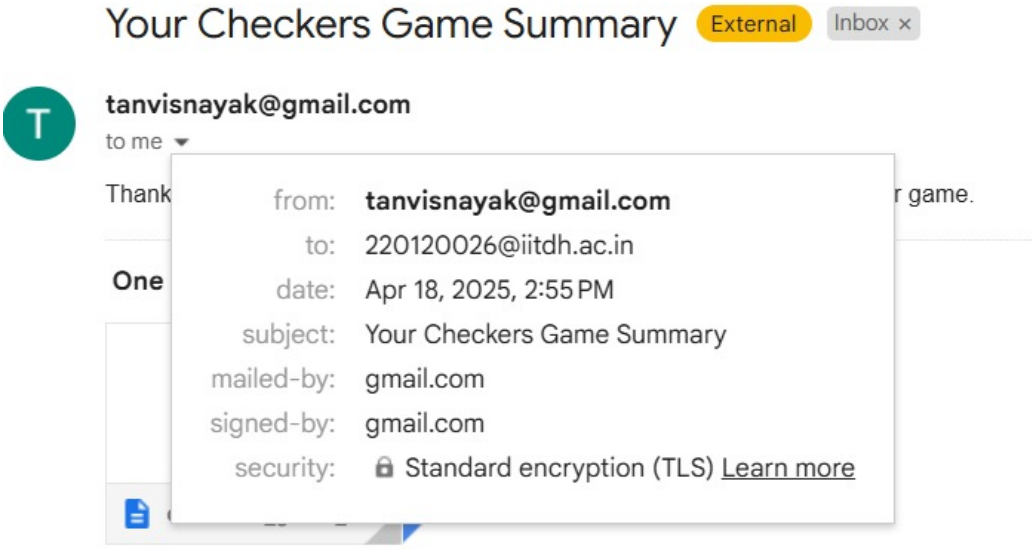

Figure 4: Email received after a completed game where one player won. Note the sender address (system email), recipient address (player's email), clear subject line indicating game summary, and the email header details showing standard TLS encryption

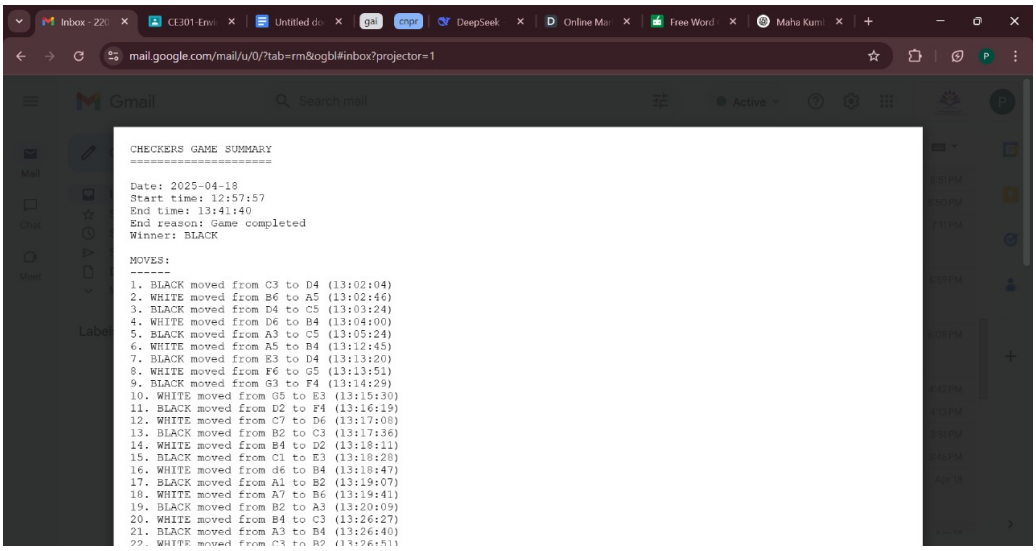

Figure 5: Email received after a completed game where one player won. Note the sender address (our system email), recipient address (player's email), clear subject line indicating game outcome, and body text with essential game information.

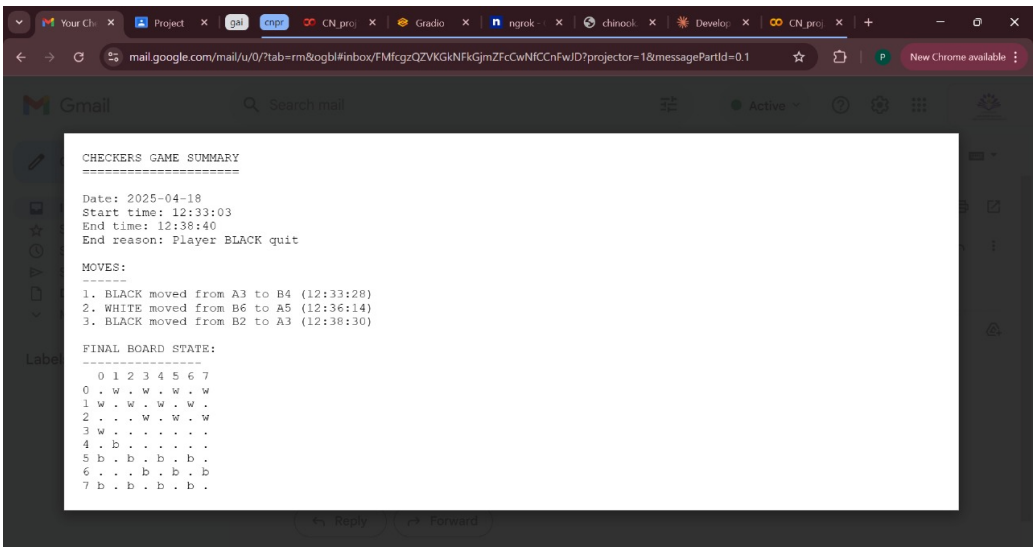

Figure 6: Email received when a game ends due to player disconnection. The email contains different messaging compared to a won game, properly identifying that one player left before completion. This demonstrates the system's ability to handle different game conclusion scenarios.

Game summary text file attached to the email, containing detailed game information including timestamps for each move, complete move history in algebraic notation, and a text representation of the final board state.

```
CHECKERS GAME SUMMARY
=====================

Date: 2025-04-18
Start time: 12:57:57
End time: 13:41:40
End reason: Game completed
Winner: BLACK

MOVES:
------
1. BLACK moved from C3 to D4 (13:02:04)
2. WHITE moved from B6 to A5 (13:02:46)
3. BLACK moved from D4 to C5 (13:03:24)
4. WHITE moved from D6 to B4 (13:04:00)
5. BLACK moved from A3 to C5 (13:05:24)
6. WHITE moved from A5 to B4 (13:12:45)
7. BLACK moved from E3 to D4 (13:13:20)
8. WHITE moved from F6 to G5 (13:13:51)
9. BLACK moved from G3 to F4 (13:14:29)
10. WHITE moved from G5 to E3 (13:15:30)
11. BLACK moved from D2 to F4 (13:16:19)
12. WHITE moved from C7 to D6 (13:17:08)
13. BLACK moved from B2 to C3 (13:17:36)
14. WHITE moved from B4 to D2 (13:18:11)
15. BLACK moved from C1 to E3 (13:18:28)
16. WHITE moved from d6 to B4 (13:18:47)
```

```
17. BLACK moved from A1 to B2 (13:19:07)
18. WHITE moved from A7 to B6 (13:19:41)
19. BLACK moved from B2 to A3 (13:20:09)
20. WHITE moved from B4 to C3 (13:26:27)
21. BLACK moved from A3 to B4 (13:26:40)
22. WHITE moved from C3 to B2 (13:26:51)
23. BLACK moved from B4 to A5 (13:27:01)
24. WHITE moved from B6 to C5 (13:27:14)
25. BLACK moved from D4 to B6 (13:28:07)
26. WHITE moved from B8 to C7 (13:28:27)
27. BLACK moved from B6 to A7 (13:28:44)
28. WHITE moved from B2 to A1 (13:29:01)
29. BLACK moved from A7 to B8 (13:29:18)
30. WHITE moved from H6 to G5 (13:31:47)
31. BLACK moved from B8 to D6 (13:32:00)
32. WHITE moved from E7 to C5 (13:32:25)
33. BLACK moved from F4 to H6 (13:32:43)
34. WHITE moved from C5 to B4 (13:33:19)
35. BLACK moved from A5 to B6 (13:33:56)
36. WHITE moved from B4 to A3 (13:34:37)
37. BLACK moved from B6 to A7 (13:34:48)
38. WHITE moved from A3 to B2 (13:35:20)
39. BLACK moved from A7 to B8 (13:35:31)
40. WHITE moved from B2 to C1 (13:35:51)
41. BLACK moved from E3 to D4 (13:36:16)
42. WHITE moved from G7 to F6 (13:36:32)
43. BLACK moved from D4 to C5 (13:36:51)
44. WHITE moved from D8 to C7 (13:37:22)
45. BLACK moved from B8 to D6 (13:37:34)
46. WHITE moved from C1 to D2 (13:37:53)
47. BLACK moved from E1 to C3 (13:38:05)
48. WHITE moved from F6 to E5 (13:38:33)
49. BLACK moved from D6 to F4 (13:38:44)
50. WHITE moved from F8 to G7 (13:39:06)
51. BLACK moved from H6 to F8 (13:39:20)
52. WHITE moved from H8 to G7 (13:39:35)
53. BLACK moved from F8 to H6 (13:39:48)
54. WHITE moved from A1 to B2 (13:40:26)
55. BLACK moved from C3 to B4 (13:40:52)
56. WHITE moved from B2 to C1 (13:41:06)
57. BLACK moved from F4 to E3 (13:41:20)
58. WHITE moved from C1 to D2 (13:41:31)
59. BLACK moved from E3 to C1 (13:41:40)

FINAL BOARD STATE:
----------------
  A B C D E F G H
8 . . . . . . . .
7 . . . . . . . .
6 . . . . . . . B
```

```
5 . . b . . . . .
4 . b . . . . . .
3 . . . . . . . .
2 . . . . . b . b
1 . . B . . . b .
```

4.3 DNS Ad-Blocking Functionality

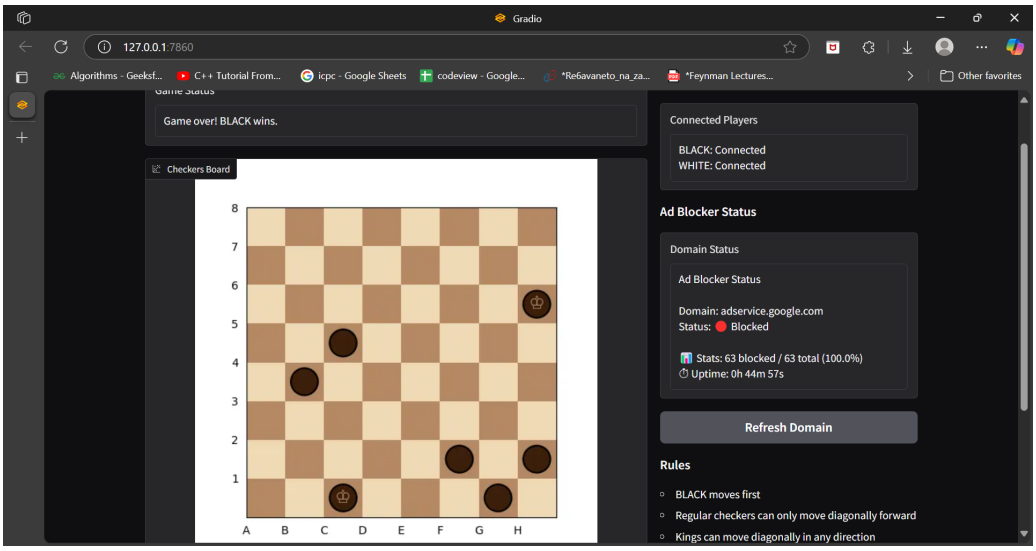

Figure 7: The Adspace section of the game interface displaying a blocked domain. When a blacklisted advertising domain is detected, the DNS resolver prevents loading of content from that domain and displays an appropriate notification to the user.

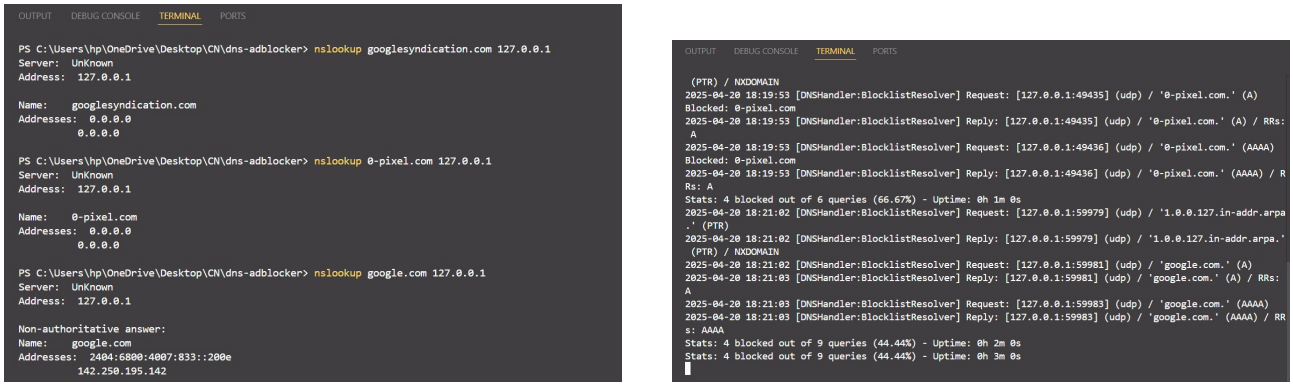

Figure 8: Terminal outputs showing nslookup commands for blocked advertising domains. The responses indicate that the domains could not be resolved, confirming that our DNS filter successfully intercepted and blocked the queries.

These observations demonstrate that our DNS-based ad-blocking mechanism functions as intended, providing effective filtering without significantly impacting network performance or user experience.

5 Demo Video

https://youtu.be/qsTc_DqKIaQ?feature=shared This is the link to the demo video hosted on youtube.

6 Setup Guide

6.1 Prerequisites

- Python 3+
- dnslib package
- requests package
- gradio package

6.2 Installation and Setup

1. Create and activate a Python virtual environment:

```
# Create virtual environment
python3 -m venv venv

# Activate virtual environment
# On Linux/macOS:
source venv/bin/activate
# On Windows:
# venv\Scripts\activate
```

2. Install required packages:

```
pip install dnslib
pip install requests
pip install gradio
```

3. Run the setup script:

```
./setup.sh
```

4. Follow the instructions displayed in the terminal.

5. Start the DNS adblocker with admin privileges:

```
sudo ./dns_adblocker.py --download --port 53
```

6. Launch the application components on different terminals:

```
python3 server_combine.py
python3 client_combine.py
```

6.3 Important Note

Before launching the application, ensure that the server email credentials(email id and app password) are included in the `email_handler.py` file within the `EmailHandler` class as shown in *Listing 6*.
Ensure that `server_bridge.py` and `email_handler.py` are located in the same directory as all other project files.

7 Future Works

The current implementation of the Checkers project successfully integrates multiplayer gaming, DNS-based ad-blocking, and SMTP-based game summaries. However, several enhancements and extensions could further improve the system:

7.1 Enhanced Game Features

- **AI Opponent Mode:** Implement an AI-based opponent for single-player mode using reinforcement learning or rule-based strategies.
- **Multiplayer Over Internet:** Extend the game to support remote players beyond localhost using NAT traversal or cloud-based matchmaking.
- **Game Replays and History:** Store game logs and allow players to review past matches via the email service.

7.2 DNS Ad-Blocker Improvements

- **Dynamic Blocklist Updates:** Automate periodic updates of blocklists without manual script execution.
- **Caching Mechanism:** Improve DNS resolution speed by caching frequently accessed domains.
- **User Customization:** Allow users to add/remove domains from blocklists via a UI.

7.3 Email Service Enhancements

- **HTML Game Summaries:** Replace plain-text emails with formatted HTML summaries, including move history and board snapshots.
- **Secure Authentication:** Implement OAuth for SMTP instead of storing plain credentials.

8 Contributions

- Maddela Vaishwitha - Frontend, Report, Email Service
- Sujata Kar - DNS Ad Blocker, Checkers Game, Report
- Vatista Kachroo - DNS Ad Blocker, Checkers Game , Report
- Prakriti Tripathi - Frontend, Checkers Game, Video Recording
- Tanvi Nayak - Email Service, Checkers Game, Report

9 Conclusion

This project successfully combines multiplayer gaming, DNS filtering, and email automation using core networking concepts. The Checkers game demonstrates real-time TCP communication, while the DNS ad-blocker and SMTP service showcase practical protocol implementations. Future upgrades could expand functionality with AI, better security, and scalability, making it a strong foundation for network-based applications.