EE129 Project 2: A Simple Web Server in Python
EE129: Computer Communication Networks

Matthew Bass
Tufts University

November 18, 2024

Introduction

This project involves the implementation of two web servers in Python:

- **Dynamic Color Selection Web Server:** This server dynamically generates an HTML form with multiple color options. Upon selection, it processes the input and responds with a styled webpage reflecting the chosen color.
- **Image Serving Web Server:** This server serves a binary image file (`rick_roll.jpg`) to clients and handles errors gracefully with a user-friendly 404 page.

The servers showcase key principles of web server design:

- Socket programming for TCP/IP communication.
- Handling HTTP requests and responses.
- Multithreading for efficient client handling.
- Error management and scalability.

Both servers dynamically configure resources to match the system's capabilities, leveraging Python's `multiprocessing` module.

Experimental Setup

General Setup

The following tools and configurations were used to implement and test the servers:

Hardware:

- Device: M1 MacBook Air.
- CPU: 8-core processor.

Software Environment:

- Operating System: macOS Ventura.
- Python Version: 3.11 (Anaconda Distribution).

Libraries Used:

- `socket` for network communication.
- `concurrent.futures` for multithreading.
- `multiprocessing` for resource-based configuration.
- `logging` for debugging and server monitoring.

Port Assignments:

- Color Server: Port 6421.
- Image Server: Port 6420.

Thread Management:

- The number of worker threads (`MAX_WORKERS`) was dynamically determined:

```
MAX_WORKERS = multiprocessing.cpu_count()
```

Both servers allowed a connection backlog of 100 requests for testing.

How to Run the Servers

To run the servers, follow these steps:

Dynamic Color Selection Web Server:

1. Navigate to the directory containing the server script (`color_server.py`).
2. Run the server using the following command:
   ```
   python color_server.py
   ```
3. Access the server in a web browser by navigating to `http://localhost:6421`.

Image Serving Web Server:

1. Navigate to the directory containing the server script (`image_server.py`).
2. Ensure the file `rick_roll.jpg` is in the same directory as the server script.
3. Run the server using the following command:
   ```
   python image_server.py
   ```
4. Access the server in a web browser by navigating to `http://localhost:6420`.

Shared Components

Both servers share a common core infrastructure for socket initialization, thread management, and logging. The foundational components are defined as follows:

```
from socket import *
import sys
import logging
from concurrent.futures import ThreadPoolExecutor
import multiprocessing

PORT_NUMBER = 6421  # Different for each server
MAX_WORKERS = multiprocessing.cpu_count()  # Matches system CPU cores
LISTEN_BACKLOG = 100  # Size of the connection queue
```

Component Details

- **CPU-based Thread Pooling:** `MAX_WORKERS` dynamically matches the system's CPU cores for optimal parallelism.
- **Connection Backlog:** `LISTEN_BACKLOG = 100` ensures the server can handle traffic bursts without overloading resources.
- **Cross-Platform Support:** All components are compatible across macOS, Windows, and Linux.

Logging System Implementation

```
logging.basicConfig(
    level=logging.INFO,
    format='[%(threadName)s] - %(message)s'
)
```

Key Features:

- Thread Identification: Clearly tracks which thread handles each request.
- Error Detection: Captures and logs error conditions for debugging.
- Performance Monitoring: Logs request and response times for optimization.
- Request Tracking: Monitors all client connections for traceability.
- Development Assistance: Simplifies debugging through detailed logs.

Color Server Detailed Analysis

Color Management System

```
COLORS = {
    "red": "#FF0000", "green": "#00FF00", "blue": "#0000FF",
    "orange": "#FFA500", "pink": "#FFC0CB", "cyan": "#00FFFF",
    "magenta": "#FF00FF", "brown": "#7B3F00"
}
```

Advantages:

- Ease of Extension: New colors can be added simply by appending to the dictionary.
- Dynamic Updates: HTML forms update dynamically based on dictionary content.
- Validation: Ensures valid user selections through dictionary lookups.
- Efficiency: Maps colors directly to their hex codes for faster processing.
- Flexibility: Supports multiple use cases, including error handling for invalid colors.

HTML Generation System

```
def create_html(title, body, status="200 OK"):
    return (
        f"HTTP/1.1 {status}\r\nContent-Type: text/html\r\n\r\n"
        f"<html><head><title>{title}</title></head><body>{body}</body></html>
    )
```

Design Considerations:

- Consistency: Maintains uniform HTTP headers and HTML structure across responses.
- MIME Support: Specifies `Content-Type:  text/html` for browser compatibility.
- Reusability: Simplifies response creation for success, error, and dynamic content.
- Flexibility: Easily adapts to different status codes and response bodies.
- Clean Code: Encapsulates repetitive response logic in a single function.

Form Generation

```
form = ("<h1>Choose your color</h1><form action='/color'><select name='color'
        + "".join([f"<option value='{color}'>{color.title()}</option>" for co
        + "</select><input type='submit' value='Submit'></form>")
```

Key Features:

- Dynamic Option Creation: Iterates over the COLORS dictionary to generate `<option>` elements.
- Readable Formatting: Uses `.title()` for user-friendly capitalization.
- Interactive Interface: Provides a dropdown menu for seamless user interaction.
- Error Minimization: Automatically validates inputs through predefined options.
- Scalability: Adapts dynamically to changes in the COLORS dictionary.

Image Server Detailed Analysis

Binary File Operations

```
try:
    with open(filename, "rb") as f:
        outputdata = f.read()
    connection_socket.send("HTTP/1.1 200 OK\r\nContent-Type: image/jpeg\r\n\r
    connection_socket.sendall(outputdata)
```

Design Considerations:

- **Binary Mode Reading:** Ensures data integrity for non-text files by reading the image as raw binary data.
- **MIME Specification:** Sets `Content-Type: image/jpeg` in the HTTP response headers to ensure proper handling and rendering of the image by the client's browser.
- **Complete Transmission:** Uses `sendall` to guarantee delivery of the entire image data in a single operation, preventing issues where partial data might cause unintended behaviors.
- **Avoiding Downloads:** Sending binary data in smaller chunks (e.g., within a loop) can cause browsers to misinterpret the response and trigger a download instead of rendering the image inline. `sendall` ensures the entire response is transmitted correctly and interpreted as intended.
- **Resource Safety:** Automatically releases file resources via context management, ensuring that open file descriptors are cleaned up.
- **Scalability:** Handles large binary files without performance degradation, maintaining efficient transmission regardless of file size.

Error Management System

```
except IOError:
    connection_socket.send("HTTP/1.1 404 Not Found\r\n\r\n".encode())
    connection_socket.send(
        "<html><head></head><body><h1>404 Image Not Found</h1></body></html>\
    )
```

Key Features:

- HTTP Compliance: Responds with the appropriate `404 Not Found` status code.
- User-Friendly Messaging: Displays a clear and concise error page.
- Graceful Recovery: Handles missing files without server crashes.
- Resource Cleanup: Ensures sockets are properly closed even during errors.
- Logging Support: Captures error details for debugging and monitoring.

Thread Pool Implementation Details

```
with ThreadPoolExecutor(max_workers=MAX_WORKERS) as executor:
    while True:
        connection_socket, client_address = server_socket.accept()
        executor.submit(handle_client, connection_socket, client_address)
```

Advantages:

- Concurrent Handling: Manages multiple clients simultaneously without delays.
- Automatic Scaling: Adjusts to system resources dynamically.
- Queuing: Queues excess connections until threads become available.
- Load Balancing: Distributes workload evenly among threads.
- Thread Safety: Prevents resource contention during concurrent operations.

Results

Dynamic Color Selection Web Server `simple_color_web_server.py`

Functional Outcomes:

- The server successfully generated a dynamic HTML form with multiple color options.
- Processed user selections accurately and returned styled responses reflecting the chosen color.

- Handled invalid paths gracefully, serving a 404 error page.

Observed Behavior:

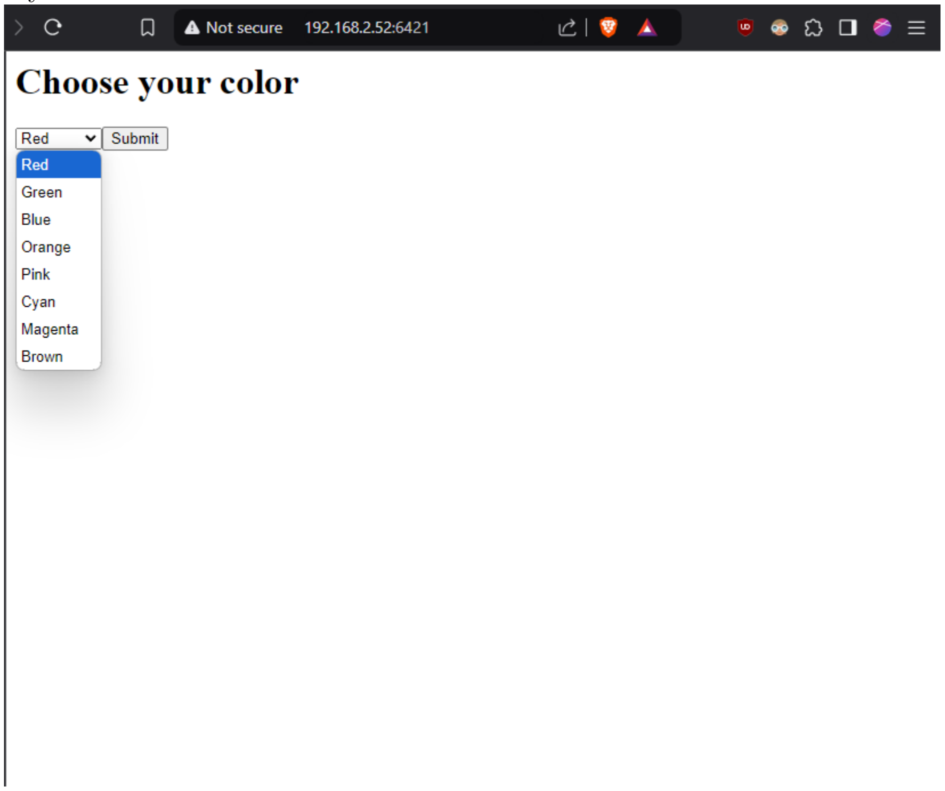
- The form dynamically updated based on the `COLORS` dictionary, ensuring extensibility and user-friendly interaction.
- Styled response pages correctly displayed the chosen color with appropriate formatting and CSS styling.
- Concurrent requests were handled efficiently without delays or errors, thanks to the thread pool.

Testing Details:

- The server was run on the M1 MacBook Air with IP `192.168.2.52`.
- The server was accessed from another PC on the same LAN, a Windows desktop with IP `192.168.2.120`.

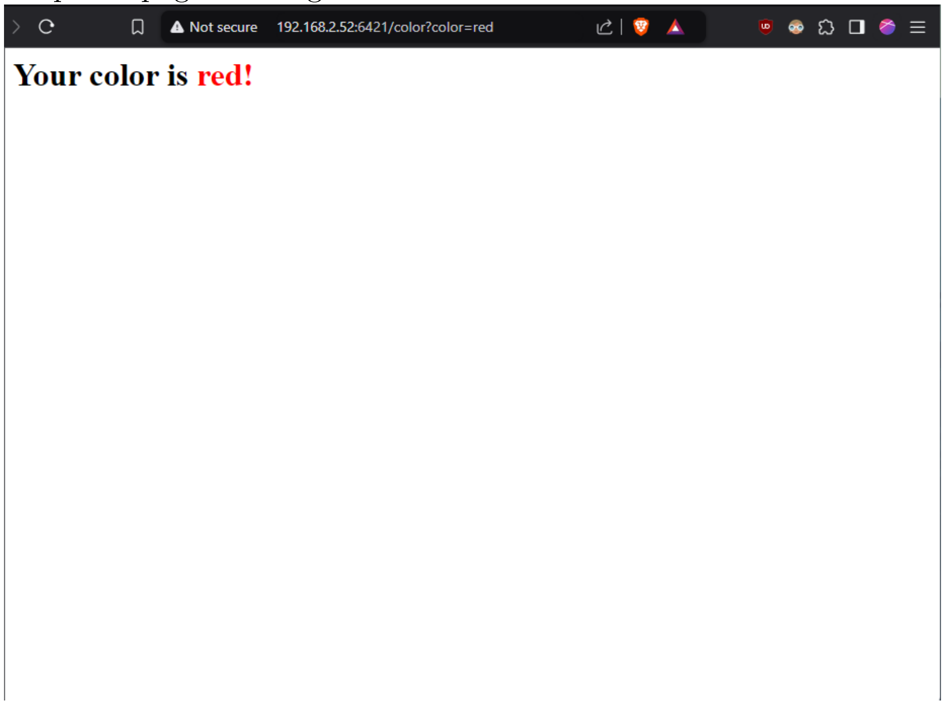
Screenshots:

- Dynamic color selection form:



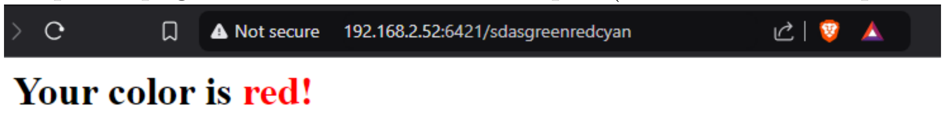
Choose your color

- Response page showing the color "red":



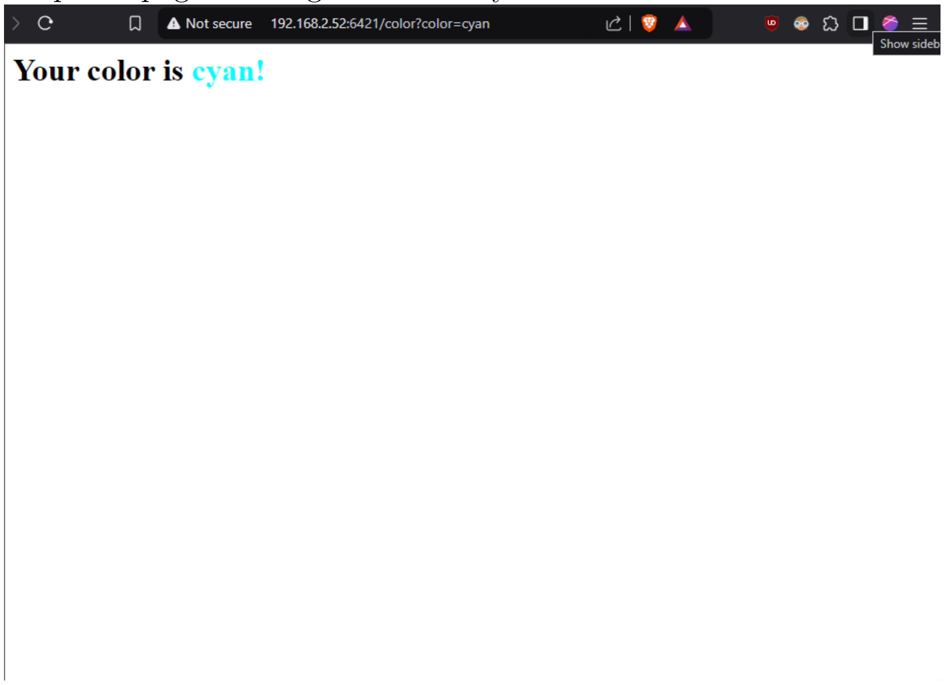
Your color is red!

- Response page with malformed "red" input (shows how red is prioritized):



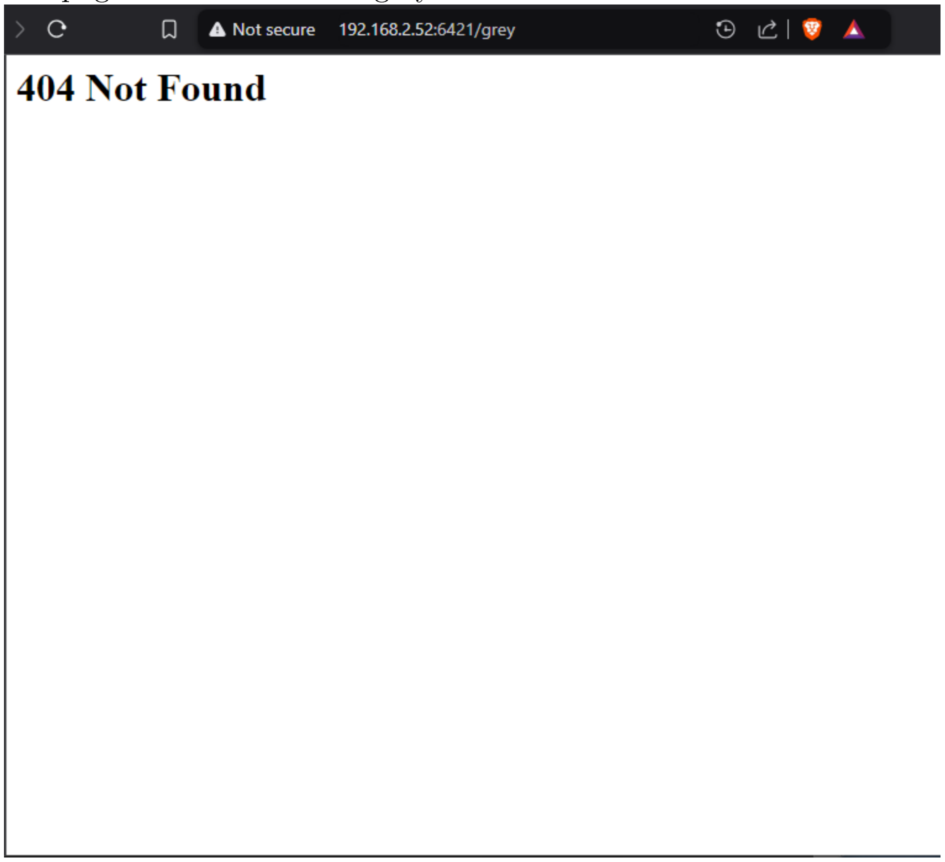
Your color is red!

- Response page showing the color "cyan":



Your color is cyan!

- 404 page for invalid color "grey":



404 Not Found

Image Serving Web Server `simple_image_web_server.py`

Functional Outcomes:

- The server reliably delivered the `rick_roll.jpg` image to clients upon valid requests.
- Returned a user-friendly 404 error page for invalid paths or missing files.
- Managed concurrent image requests seamlessly, with no noticeable performance degradation.

Observed Behavior:

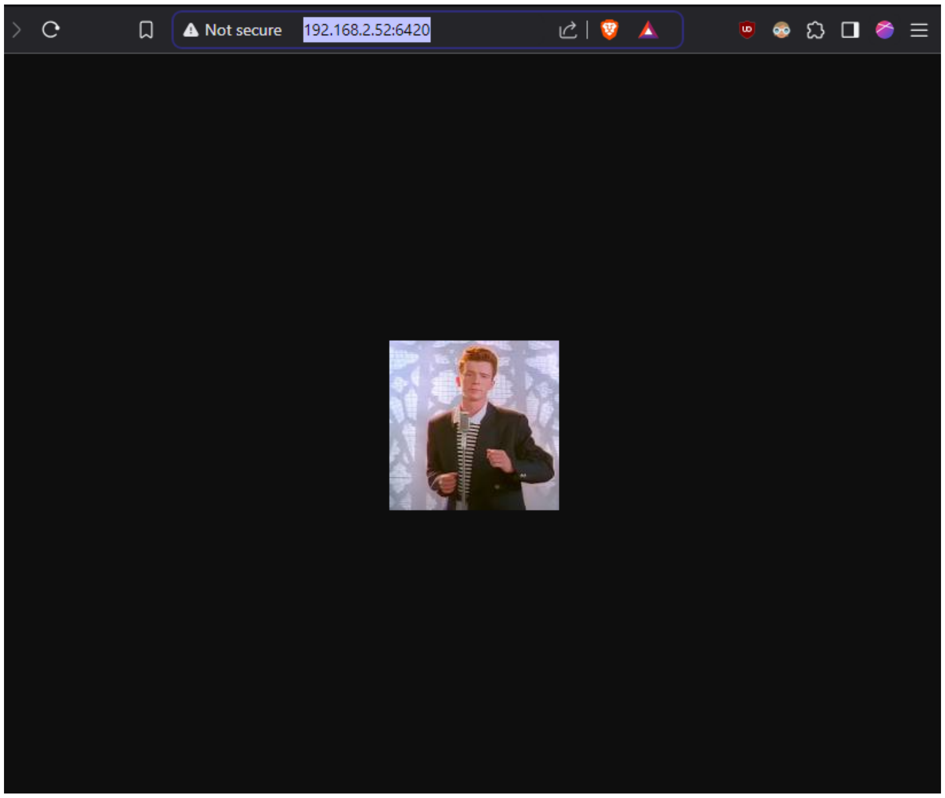
- The server transmitted binary data without loss, using proper MIME type headers (`Content-Type: image/jpeg`) to ensure compatibility with browsers.
- Concurrent requests were distributed across threads effectively, leveraging the `ThreadPoolExecutor` for optimal performance.
- Error responses included user-friendly HTML error messages, maintaining a consistent user experience even when requests failed.

Testing Details:

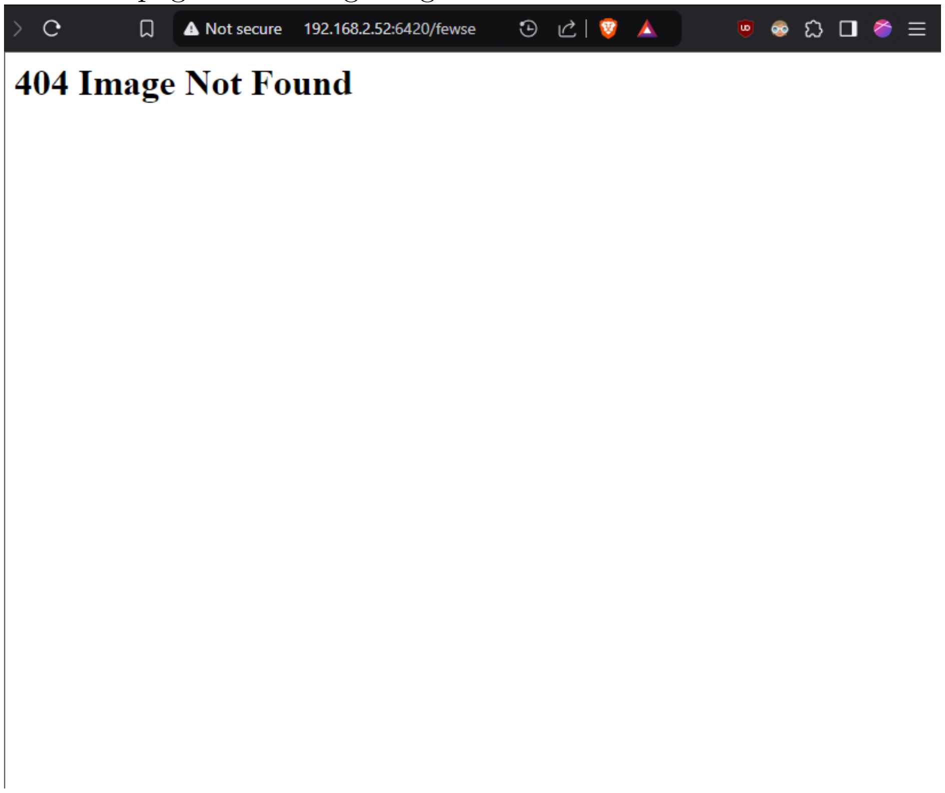
- The server was run on the M1 MacBook Air with IP `192.168.2.52`.
- The server was accessed from another PC on the same LAN, a Windows desktop with IP `192.168.2.120`.

Screenshots:

- Successfully delivered image (`rick_roll.jpg`):



- 404 error page for missing image:



404 Image Not Found

Extra Credit Fulfillment

Program Length

Both programs were under 100 lines by using modular functions, dynamic dictionaries, and efficient logging.

Execution

The programs include the `if __name__ == "__main__"` structure for straightforward execution:

```
if __name__ == "__main__":
    server_socket = initialize_server(PORT_NUMBER)
    run_server(server_socket)
```

Port Management

Implemented `SO_REUSEADDR` to reuse ports during rapid restarts:

```
server_socket.setsockopt(SOL_SOCKET, SO_REUSEADDR, 1)
```

Access Across LAN

- Both servers were tested from a different computer on the same LAN, fulfilling the requirement to access the web server via its IP address.
- The client machine (Windows desktop at `192.168.2.120`) successfully connected to the servers hosted on the M1 MacBook Air at `192.168.2.52`.

Additional Features

- Dynamic HTML generation. - Comprehensive error handling. - Binary file transfer with MIME type support.

Conclusion

The project demonstrated the successful implementation of two specialized web servers using Python. The **Dynamic Color Selection Web Server** `simple_color_web_server.py` effectively showcased dynamic content generation, using a dictionary-based system for extensibility and a reusable HTML generation function for flexibility. The **Image Serving Web Server** `simple_image_web_server.py` handled binary data efficiently, providing reliable file transfers and robust error handling. Both servers incorporated multithreading through the `ThreadPoolExecutor`, ensuring scalability and responsiveness even under concurrent loads.

The servers achieved the functional goals outlined in the introduction, highlighting the application of socket programming, HTTP communication, and multithreaded server design.

Challenges such as managing concurrent connections, handling malformed requests, and implementing efficient binary file transfers were resolved through thoughtful design choices, including dynamic thread allocation and comprehensive logging.

This project reinforced concepts of network programming, demonstrating how to build scalable and robust server applications. Key learnings included dynamic HTML response creation, binary file handling, multithreaded programming, and user-friendly error management. The servers also emphasize the importance of modular design, enabling reusable and extensible components for future enhancements.

Appendix

A. simple_color_web_server.py

```
"""
simple_color_web_server.py
This server listens for incoming TCP connections and serves a color selection form.
Uses ThreadPoolExecutor to handle client connections efficiently.
"""
from socket import *
import sys
import logging
from concurrent.futures import ThreadPoolExecutor
import multiprocessing

PORT_NUMBER = 6421  # Port to listen on
MAX_WORKERS = multiprocessing.cpu_count()  # Number of worker threads in the pool (I hav
LISTEN_BACKLOG = 100  # Number of connections to queue
COLORS = {"red": "#FF0000", "green": "#00FF00", "blue": "#0000FF", "orange": "#FFA500",

# Configure logging
logging.basicConfig(
    level=logging.INFO,
    format='[%(threadName)s] %(message)s'
)

def create_html(title, body, status="200 OK"):
    """
    Create HTML response with given title, body, and status.
    """
    return (
        f"HTTP/1.1 {status}\r\nContent-Type: text/html\r\n\r\n"
        f"<html><head><title>{title}</title></head><body>{body}</body></html>"
    )

def handle_client(connection_socket, client_address):
    """
    Handle the client connection.
    """
    try:
        message = connection_socket.recv(1024).decode()
        if not message:
            return
        logging.info(f"Processing request from {client_address}")
        path = message.split()[1]
        try:
            if path == "/":
                # Serve the color selection form
                form = ("<h1>Choose your color</h1><form action='/color'><select name='c
                        + "".join([f"<option value='{color}'>{color.title()}</option>" f
                        + "</select><input type='submit' value='Submit'></form>")
                response = create_html("Color Form", form)
            else:
                # Handle color selection
                for color in COLORS:
                    if color in path:
                        body = f"<h1>Your color is <span style='color:{COLORS[color]};'>
                        response = create_html(f"{color.title()} Color", body)
                        break
                else:
                    response = create_html( title="404 Not Found",body="<h1>404 Not Foun
            connection_socket.send(response.encode())
            logging.info(f"Sent response to {client_address}")
        except IOError:
            not_found = create_html(title="",body="<h1>Not Found</h1>",status="404 Not F
            connection_socket.send(not_found.encode())
            logging.info(f"Error response sent to {client_address}")
    except Exception as e:
        logging.error(f"Error handling client {client_address}: {e}")
    finally:
        connection_socket.close()
        logging.info(f"Closed connection with {client_address}")

def main():
    server_socket = socket(AF_INET, SOCK_STREAM)
    server_socket.setsockopt(SOL_SOCKET, SO_REUSEADDR, 1)
    try:
        server_socket.bind(("", PORT_NUMBER))
        server_socket.listen(LISTEN_BACKLOG)
        logging.info(f"Server started on port {PORT_NUMBER}")
        logging.info(f"Configured with:")
        logging.info(f"- Max worker threads: {MAX_WORKERS}")
        logging.info(f"- Connection backlog: {LISTEN_BACKLOG}")
        logging.info("Server ready to accept connections")
        with ThreadPoolExecutor(max_workers=MAX_WORKERS) as executor:
            while True:
                try:
                    connection_socket, client_address = server_socket.accept() # Accept
                    logging.info(f"Accepted connection from {client_address}")
                    executor.submit(handle_client, connection_socket, client_address) #
                except Exception as e:
                    logging.error(f"Error accepting connection: {e}")
                    continue
    except KeyboardInterrupt:
        logging.info("\nShutting down server...")
    except Exception as e:
        logging.error(f"Server error: {e}")
    finally:
        server_socket.close()
        sys.exit(0)

if __name__ == "__main__":
    main()
```

B. simple_image_web_server.py

```
"""
simple_image_web_server.py
This server listens for incoming TCP connections with a configurable backlog.
Uses ThreadPoolExecutor to handle client connections efficiently.
"""
from socket import *
import sys
import logging
from concurrent.futures import ThreadPoolExecutor
import multiprocessing

PORT_NUMBER = 6420  # Port to listen on
MAX_WORKERS = multiprocessing.cpu_count()  # Number of worker threads in the pool (I hav
LISTEN_BACKLOG = 100  # Number of connections to queue
IMAGE_FILE = "rick_roll.jpg" # File to serve

# Configure logging
logging.basicConfig(
    level=logging.INFO,
    format='[%(threadName)s] %(message)s'
)

def handle_client(connection_socket, client_address):
    """
    Handle the client connection.
    """
    try:
        message = connection_socket.recv(1024).decode()
        if not message:
            return
        logging.info(f"Processing request from {client_address}")

        filename = "rick_roll.jpg"
        try:
            with open(filename, "rb") as f:
                outputdata = f.read()
            connection_socket.send("HTTP/1.1 200 OK\r\nContent-Type: image/jpeg\r\n\r\n"
            connection_socket.sendall(outputdata)
            logging.info(f"Sent file to {client_address}")

        except IOError:
            connection_socket.send("HTTP/1.1 404 Not Found\r\n\r\n".encode())
            connection_socket.send(
                "<html><head></head><body><h1>404 Image Not Found</h1></body></html>\r\n
            logging.info(f"File not found error for {client_address}")

    except Exception as e:
        logging.error(f"Error handling client {client_address}: {e}")
    finally:
        connection_socket.close()
        logging.info(f"Closed connection with {client_address}")

def main():
    server_socket = socket(AF_INET, SOCK_STREAM)
    server_socket.setsockopt(SOL_SOCKET, SO_REUSEADDR, 1)

    try:
        server_socket.bind(("", PORT_NUMBER))
        server_socket.listen(LISTEN_BACKLOG)
        logging.info(f"Server started on port {PORT_NUMBER}")
        logging.info(f"Configured with:")
        logging.info(f"- Max worker threads: {MAX_WORKERS}")
        logging.info(f"- Connection backlog: {LISTEN_BACKLOG}")
        logging.info("Server ready to accept connections")

        with ThreadPoolExecutor(max_workers=MAX_WORKERS) as executor:
            while True:
                try:
                    # Accept will pull from the backlog queue
                    connection_socket, client_address = server_socket.accept()
                    logging.info(f"Accepted connection from {client_address}")

                    # Submit to thread pool - if pool is full, task will queue internall
                    executor.submit(handle_client, connection_socket, client_address)

                except Exception as e:
                    logging.error(f"Error accepting connection: {e}")
                    continue

    except KeyboardInterrupt:
        logging.info("\nShutting down server...")
    except Exception as e:
        logging.error(f"Server error: {e}")
    finally:
        server_socket.close()
        sys.exit(0)

if __name__ == "__main__":
```

```
main()
```