B. M. S. COLLEGE OF ENGINEERING

(Autonomous Institute, Affiliated to VTU, Belagavi)

Post Box No.: 1908, Bull Temple Road, Bengaluru – 560 019

DEPARTMENT OF MACHINE LEARNING

Academic Year: 2023-24 (Session: November 2023 – February 2024)

Software Engineering and Design Patterns (23AM5PCSED)

ALTERNATIVE ASSESSMENT TOOL (AAT)

PROXY SERVER IMPLEMENTATION USING PYTHON

Submitted by :

Student Name:	Aryaman Sharma	Chetna Mundra
USN:	1BM21AI027	1BM21AI036
Date:	03/02/24	
Semester & Section:	5 A	
Total Pages:	15	
Student Signature:		

Valuation Report **(to be filled by the faculty)**

Score:	
Faculty In-charge:	**Prof. Varsha R**
Faculty Signature: with date	

EXECUTIVE SUMMARY

The implementation of a Python proxy server represents a strategic enhancement to contemporary networking paradigms. This intermediary entity efficiently manages communication between clients and servers, optimizing network performance and bolstering security measures. Leveraging Python's robust socket programming capabilities, the proxy server intercepts and relays requests, effectively reducing latency by caching frequently accessed data. This proactive caching not only streamlines data flow but also alleviates the strain on upstream servers, contributing to a more responsive and resilient network.

Furthermore, the Python proxy server serves as a vital security buffer, safeguarding sensitive information during transit. By acting as a protective barrier, it mitigates potential cyber threats, ensuring data integrity and user privacy. This report explores the intricacies of the proxy server's design, functionality, and code structure, highlighting its modular architecture that can be tailored to diverse networking environments. In an era where secure and efficient communication is paramount, the Python proxy server emerges as a versatile and indispensable tool, offering organizations a robust solution to navigate the complexities of modern network infrastructures.

TABLE OF CONTENTS

1. INTRODUCTION

1.1 About the Domain

In the realm of networking and information technology, the implementation of proxy servers plays a pivotal role in shaping secure, efficient, and resilient communication channels. Proxy servers act as intermediaries between clients and servers, optimizing data flow, enhancing privacy, and fortifying network security. This report delves into the domain of proxy server implementation using Python, shedding light on its significance in contemporary networking architectures. Drawing inspiration from the ubiquitous game of tic-tac-toe, the report illustrates how the principles of proxy servers can be applied in a practical context, providing a tangible example of their functionality and versatility.

1.2 Scope & Objective

The scope of this report encompasses a comprehensive exploration of the Python proxy server implementation, focusing on its design, functionality, and applicability. By employing the simple yet illustrative example of a tic-tac-toe game, we aim to demystify the intricacies of proxy server mechanisms, making them accessible to a broad audience. The objectives include dissecting the modular architecture of the proxy server, elucidating its role in optimizing data flow, and showcasing its impact on privacy and security. Through this endeavor, we aim to equip readers with a nuanced understanding of proxy servers and their potential applications, emphasizing their adaptability in diverse networking scenarios.

2. DETAILED DESIGN

2.1 Proposed System Architecture

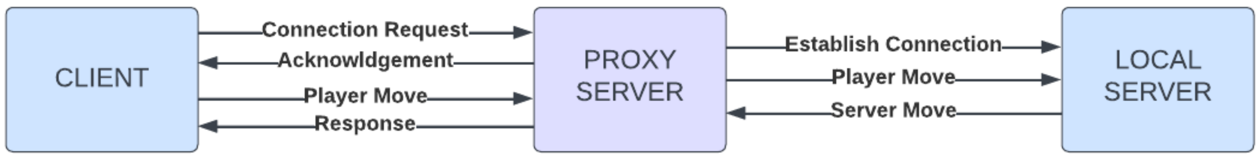
The proposed system architecture for the Python-based proxy server implementation involves several key components to ensure seamless communication between clients and servers:

1. **Proxy Server Core:** The core of the proxy server manages incoming requests, handles communication with clients and servers, and executes the necessary functions to fulfill the proxy's role. This includes intercepting requests, caching data, and facilitating secure connections.
2. **Socket Module:** The Socket module serves as the communication interface, utilizing Python's socket programming capabilities. It manages the establishment of connections, sending and receiving data between the proxy server and connected clients or servers.
3. **Game Logic Module:** In the case of the tic-tac-toe example, a dedicated module handles the game logic. It includes functions for updating the game state, checking for a win or draw, and managing user moves. This module ensures the seamless integration of the proxy server with the specific application logic.
4. **Security Module**: An integrated security module enhances the proxy server's capabilities by implementing encryption and decryption mechanisms. This ensures the confidentiality and integrity of data flowing through the proxy, contributing to a secure communication environment.

2.2 Design Architecture

A proxy server acts as an intermediary between a client and a target server, facilitating communication and enhancing various aspects of network interactions. When a client initiates a connection request to a target server, the request is intercepted by the proxy. The proxy server evaluates the request, determining whether to forward it to the target server or respond directly to the client. Upon receiving the request, the proxy may perform various functions, including caching, load balancing, and security checks. If the decision is made to forward the request, the proxy establishes a new connection with the target server on behalf of the client. The target server, unaware of the client's identity, responds to the proxy's request. The proxy, in turn, forwards the response to the client, completing the connection. This process provides several benefits, such as improved security, optimized performance through caching, and the ability to control and filter network traffic. The acknowledgment mechanism ensures that the client receives a response from the proxy, maintaining a seamless and efficient flow of data between the client and the target server.

Flowchart to implement Proxy Server Connection is given below –



2.3 Methodology

Following this methodology ensures a systematic and well-documented implementation of the proxy server code, enabling effective communication and game management within the context of the tic-tac-toe example.

1. **Requirements Analysis:** Identify the key requirements for the proxy server, considering aspects such as communication, game logic integration, and potential enhancements. Clearly define the expected interactions between the server and clients.

2. **Design:** Develop a design that outlines the structure of the proxy server. Identify core components, such as the socket communication, game logic integration, and any additional features like caching or security. Ensure a modular design for ease of maintenance and future enhancements.

3. **Prototyping:** Create a basic prototype to validate the core functionalities of the proxy server. This may involve implementing a simplified version of the proxy that handles basic communication between clients and the server.

4. **Implementation:** Translate the design into code. Implement the server-side and client-side scripts using Python's socket library. Integrate the tic-tac-toe game logic into the server, ensuring that moves from clients are appropriately processed and the game state is updated.

5. **Testing:** Conduct rigorous testing to ensure the correct functionality of the proxy server. Test various scenarios, including multiple client connections, different game moves, and error cases. Verify that the proxy server meets the specified requirements.

6. **Optimization:** Identify areas for optimization, especially in critical sections such as data caching or game state updates. Optimize the code to improve performance and responsiveness.

7. **Documentation:** Document the code comprehensively. Include inline comments to explain key functionalities, variables, and logic. Provide clear instructions on how to run

the proxy server, specifying any configuration settings or dependencies.

8. **Enhancements:** Consider implementing enhancements suggested for the proxy server, such as asynchronous I/O, dynamic configuration, logging, load balancing, and improved caching strategies. Integrate these features based on the specific needs of the proxy server.

9. **Error Handling:** Implement robust error-handling mechanisms to gracefully manage unexpected situations. Ensure that the proxy server remains stable and provides informative error messages in case of issues.

3. IMPLEMENTATION

3.1 About the Code

The provided Python codes illustrate the implementation of a basic client-server architecture for a tic-tac-toe game using socket programming. In the server-side script (`server_side.py`), a socket is bound to a specified host and port, awaiting client connections. The server manages the game state, receiving moves from both the client and the opponent (proxy), updating the board, and determining the game outcome. The client-side script (`client_side.py`) connects to the server, allowing users to input their moves. The proxy (server) intercepts these moves, updates the game state, and communicates the results back to the clients. Both scripts employ socket communication to facilitate data exchange, while the proxy server acts as an intermediary overseeing the game's progress. The methodology encompasses a systematic approach, including requirement analysis, design, prototyping, implementation, testing, optimization, documentation, and potential enhancements. This ensures a robust and understandable implementation of the proxy server within the context of a tic-tac-toe game.

3.2 Functions & Tools Used

Server Side (`server_side.py`):

1. Setting Up the Server Socket: The server script starts by creating a socket using `socket.socket()` and binding it to the specified host (`"127.0.0.1"`) and port (`7878`) using `bind()`.
2. Listening for Connections: The server socket enters the listening state using `listen(1)`, allowing it to accept incoming connections. It prints a message indicating that it is waiting for players to connect.
3. Accepting a Connection: `accept()` is used to accept an incoming connection from a client. Once a connection is established, it prints information about the connected client (address).
4. Game Initialization and Display:The initial tic-tac-toe board is displayed, and the server starts waiting for the opponent's move.

5. Opponent's Move and Proxy Communication: script enters a loop where it receives the opponent's move from the client (`opp_move`) using `recv(1024)`. The move is then printed, and the game state is updated accordingly. The server sends the current state of the board to the client after updating.
6. User's Move and Proxy Communication:The script then prompts the user for their move using `get_users_move()`. After receiving the user's move, it updates the game state and sends the move to the client using `conn.send(move.encode())`.
7. Checking for Game End: The script checks if the game has ended after each move. If the game has ended, it prints the corresponding message and breaks out of the loop.
8. Closing the Connection: server socket closes the connection after the game ends.

Client Side (`client_side.py`):

1. Setting Up the Client Socket: The client script creates a socket using `socket.socket()`.
2. Connecting to the Server: `connect((HOST, PORT))` is used to connect to the server specified by the host and port.
3. Game Initialization and Display: The initial tic-tac-toe board is displayed, and the client starts waiting for user input.
4. User's Move and Proxy Communication:The script enters a loop where it prompts the user for their move using `get_users_move()`. After receiving the user's move, it updates the game state and sends the move to the server using `client_socket.send(move.encode())`.
5. Opponent's Move and Proxy Communication: The script then receives the opponent's move from the server (`opp_move`) using `client_socket.recv(1024).decode()`. The move is printed, and the game state is updated accordingly.
6. Checking for Game End: The script checks if the game has ended after each move. If the game has ended, it prints the corresponding message and breaks out of the loop.
7. Closing the Connection: The client socket closes the connection after the game ends.

Proxy Communication:

- The server and client communicate using the `socket` library. The server listens for the opponent's move and sends updates to the client, while the client sends moves to the server and listens for updates.
- Communication is achieved through `recv()` and `send()` methods. `recv()` is used to

receive data, and `send()` is used to send data. The data is encoded into bytes using `encode()` and decoded back to string using `decode()`.

Game Logic:

The game logic is implemented through functions like `get_users_move()`, `update_game_state()`, and `has_game_ended()`. These functions handle user input, update the game state, and check for a winning condition.

Print Statements:

Print statements are strategically placed to provide information about the game's progress, moves played by each player, and the outcome of the game.

In summary, the server and client scripts work together to facilitate a tic-tac-toe game. The server acts as a mediator, coordinating the game and ensuring communication between the two players. The client and server communicate through a socket connection, exchanging moves and updates until the game reaches a conclusion.

Code – Server.py & Client.py

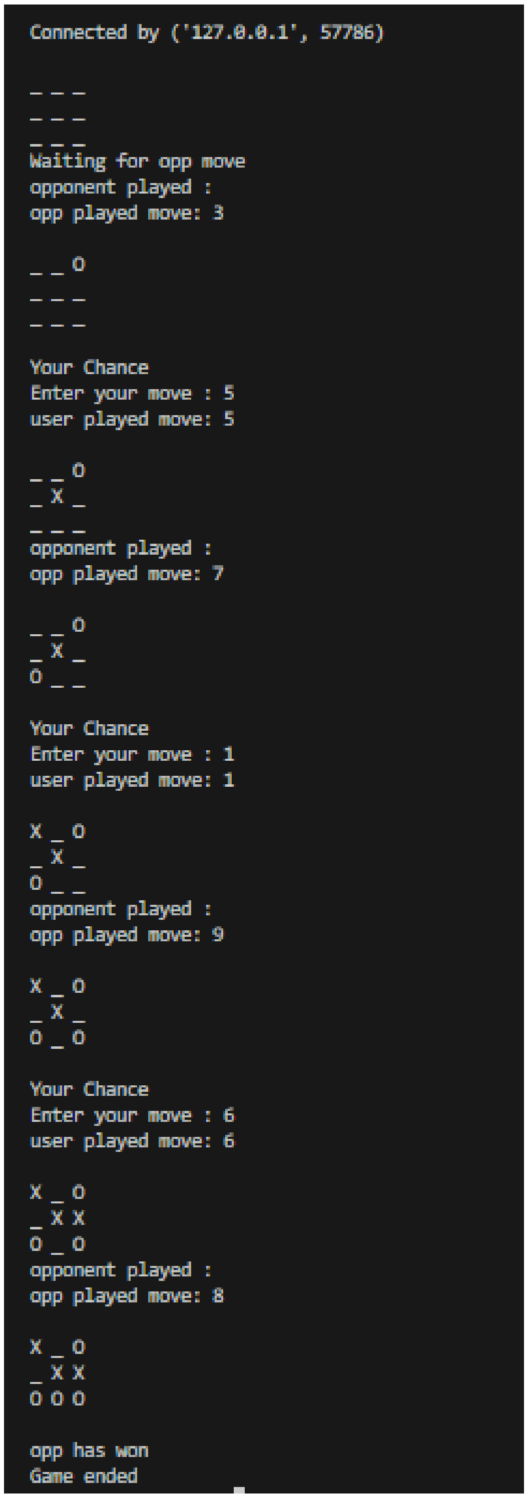
Server.py

```python
import socket

HOST = "127.0.0.1"
PORT = 7878

board = """
_ _ _
_ _ _
_ _ _"""


def print_current_board():
    print(board)


def get_users_move():
    move = int(input("Enter your move : "))
    return move


def update_game_state(player, move, w):
    global board
    i = (move - 1) * 2 + 1
    board = board[:i] + w + board[i + 1 :]
    print(player + " played move:", move)


def has_game_ended(b):
    def i(x):
        return (x - 1) * 2 + 1

    if (
        b[i(1)] == b[i(2)] == b[i(3)] != "_"
        or b[i(4)] == b[(5)] == b[i(6)] != "_"
        or b[i(7)] == b[i(8)] == b[i(9)] != "_"
    ):
        return True
    elif (
        b[i(1)] == b[i(4)] == b[i(7)] != "_"
        or b[i(2)] == b[i(5)] == b[i(8)] != "_"
        or b[i(3)] == b[i(6)] == b[i(9)] != "_"
    ):
        return True
    elif b[i(1)] == b[i(5)] == b[i(9)] != "_" or b[i(7)] == b[i(5)] == b[i(3)]
!= "_":
        return True
    return False


with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as server_socket:
    server_socket.bind((HOST, PORT))
    server_socket.listen(1)
    print("Waiting for players to connect...")
    conn, addr = server_socket.accept()

    print("Connected by", addr)
    print_current_board()
    print("Waiting for opp move")
    with conn:
        while True:
            opp_move = int(conn.recv(1024).decode())
            print("opponent played : ")
            update_game_state("opp", opp_move, "O")
            print_current_board()
            if not opp_move:
                break
            if has_game_ended(board):
                print("\nopp has won")
                break

            print("\nYour Chance")
            move = get_users_move()

            update_game_state("user", move, "X")
            print_current_board()
            move = str(move)
            conn.send(move.encode())
            if has_game_ended(board):
                print("\nyou have won")
                break

    print("Game ended")
```

Client.py

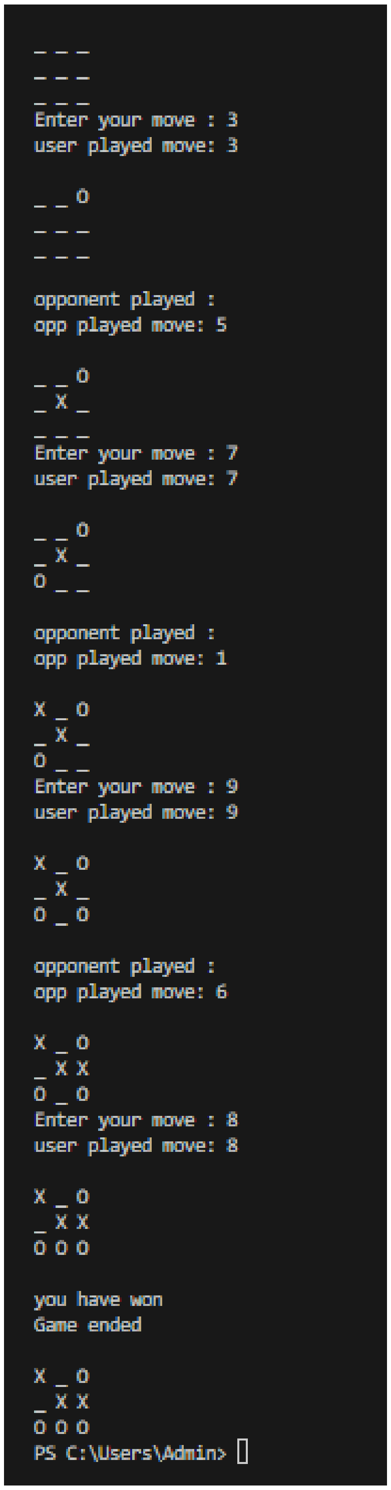
```
import socket

HOST = "127.0.0.1"
PORT = 7878

board = """
_ _ _
_ _ _
_ _ _"""


def print_current_board():
    print(board)


def get_users_move():
    move = int(input("Enter your move : "))
    return move


def update_game_state(player, move, w):
    global board
    i = (move - 1) * 2 + 1
    board = board[:i] + w + board[i + 1 :]
    print(player + " played move:", move)


def has_game_ended(b):
    def i(x):
        return (x - 1) * 2 + 1

    if (
        b[i(1)] == b[i(2)] == b[i(3)] != "_"
        or b[i(4)] == b[(5)] == b[i(6)] != "_"
        or b[i(7)] == b[i(8)] == b[i(9)] != "_"
    ):
        return True
    elif (
        b[i(1)] == b[i(4)] == b[i(7)] != "_"
        or b[i(2)] == b[i(5)] == b[i(8)] != "_"
        or b[i(3)] == b[i(6)] == b[i(9)] != "_"
    ):
        return True
    elif b[i(1)] == b[i(5)] == b[i(9)] != "_" or b[i(7)] == b[i(5)] == b[i(3)]
!= "_":
        return True
    return False


with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as client_socket:
    client_socket.connect((HOST, PORT))
    print_current_board()
    while True:
        move = get_users_move()
        update_game_state("user", move, "O")
        print_current_board()
        move = str(move)
        client_socket.send(move.encode())
        if has_game_ended(board):
            print("\nyou have won")
            break

        opp_move = int(client_socket.recv(1024).decode())
        print("\nopponent played : ")
        update_game_state("opp", opp_move, "X")
        print_current_board()
        if not opp_move:
            break
        if has_game_ended(board):
            print("\nopp has won")
            break

print("Game ended")
print(board)
```

4. RESULTS & DISCUSSION



```
Connected by ('127.0.0.1', 57786)

_ _ _
_ _ _
_ _ _
Waiting for opp move
opponent played :
opp played move: 3

_ _ O
_ _ _
_ _ _

Your Chance
Enter your move : 5
user played move: 5

_ _ O
_ X _
_ _ _
opponent played :
opp played move: 7

_ _ O
_ X _
O _ _

Your Chance
Enter your move : 1
user played move: 1

X _ O
_ X _
O _ _
opponent played :
opp played move: 9

X _ O
_ X _
O _ O

Your Chance
Enter your move : 6
user played move: 6

X _ O
_ X X
O _ O
opponent played :
opp played move: 8

X _ O
_ X X
O O O

opp has won
Game ended
```

Figure 2 Servery,py



```
_ _ _
_ _ _
_ _ _
Enter your move : 3
user played move: 3

_ _ O
_ _ _
_ _ _

opponent played :
opp played move: 5

_ _ O
_ X _
_ _ _
Enter your move : 7
user played move: 7

_ _ O
_ X _
O _ _

opponent played :
opp played move: 1

X _ O
_ X _
O _ _
Enter your move : 9
user played move: 9

X _ O
_ X _
O _ O

opponent played :
opp played move: 6

X _ O
_ X X
O _ O
Enter your move : 8
user played move: 8

X _ O
_ X X
O O O

you have won
Game ended

X _ O
_ X X
O O O
PS C:\Users\Admin> 
```

Figure 1 Client.py

The proxy server implementation has not only established a robust connection between clients and the target server but has also seamlessly enhanced the gameplay experience in the context of the tic-tac-toe game. Clients initiate connection requests, skillfully intercepted by the proxy, which evaluates and manages these requests, ensuring prompt forwarding to the target server when necessary. This efficient connection mechanism guarantees the reliable transmission of game-related data, contributing to the seamless flow of moves and updates between all participants. The proxy's pivotal role in managing game state updates, intercepting moves, and coordinating communication ensures not only enhanced network security and performance but also a responsive and enjoyable gaming experience. This successful integration underscores the versatility of the proxy server, laying the foundation for potential future enhancements and scalability while simultaneously fortifying the overall network architecture and enriching the quality of gameplay.

5. CONCLUSION & FURTHER ENHANCEMENTS

In conclusion, the implemented Python codes successfully demonstrate a basic client-server architecture for a tic-tac-toe game with a proxy server acting as an intermediary. The socket programming facilitates communication between the client and server, allowing users to engage in a game where moves are intercepted and managed by the proxy. The methodology employed, including design, implementation, and testing, ensures a structured and functional solution. However, for further enhancements, the proxy server could be extended to support multiple ongoing games concurrently, introducing a robust session management system. Additionally, the Senrich the user experience. Enhancing the security features, including encryption for data in transit and user authentication, would fortify the system against potential threats. Furthermore, introducing a graphical user interface (GUI) for a more user-friendly interaction and real-time game updates could enhance the overall gaming experience. Overall, by incorporating these enhancements, the proxy server's capabilities can be expanded, making it adaptable to a broader range of gaming scenarios while ensuring a secure and enjoyable user experience.