University of Science and Technology of Hanoi

Distributed System

Midterm report

HTTP over MPI

Group : *02*

Members : *Pham Hoang Viet - 23BI14452*

: *Nghiem Xuan Son - 23BI14388*

: *Vu Xuan Thai - 23BI14397*

: *Hoang Minh Quan - 23BI14371*

: *Nguyen Ngoc Quang - 23BI14376*

: *Pham Huu Minh - 23BI14302*

: *Tran Vu Cong Minh - 23BI14303*

Major : *Cyber Security*

Lecturer : *MSc. Le Nhu Chu Hiep*

Hanoi, December 07 2025

Contents

1 Introduction **3**

2 System Architecture **3**

3 MPI (Message Passing Interface) Overview **3**

4 Code Structure and Implementation **4**
4.1 Full Source Code 4
4.2 Implementation Visuals 6

5 Deployment and Usage Guideline **7**
5.1 Prerequisites 7
5.2 Compilation 8
5.3 Usage Manual 8

6 Experimental Results **8**

7 Team Contribution **9**

8 Conclusion **9**

1 Introduction

This report details an MPI (Message Passing Interface)-based system designed to send an HTTP request from a client process to a server via a proxy process. The system addresses a common scenario where a client node within an internal network (without direct internet access) needs to communicate with external web services. By utilizing a Proxy node that has internet access, the client can offload the network request via MPI messages.

2 System Architecture

The system architecture consists of two main processes communicating within a local system, with one process acting as a gateway to the external internet.

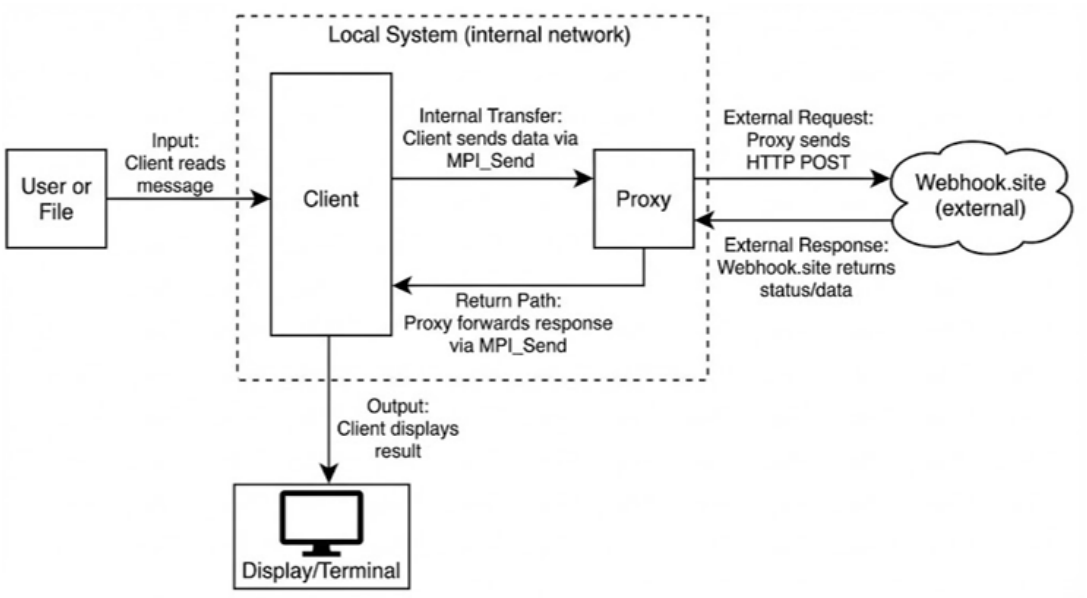

Figure 1: System Architecture Diagram: Client sends MPI message to Proxy, which forwards it as HTTP POST.

The system involves two key processes:

- **Client Process (Rank 0)**: Initiates the communication. It reads user input or file content and sends it to the proxy via MPI. It does not communicate directly with the internet.
- **Proxy Process (Rank 1)**: Acts as an intermediary. It receives the MPI message, uses `libcurl` to POST the data to an external Webhook, and relays the server's response back to the client.

3 MPI (Message Passing Interface) Overview

MPI is a standardized and portable message-passing system designed for parallel computing. This system uses MPI for inter-process communication, specifically:

- **MPI_Send**: Used by the client to send the HTTP request payload to the proxy.
- **MPI_Recv**: Used by the proxy to receive the payload, and by the client to receive the final HTTP response.

MPI Function	Purpose
`MPI_Init`	Initializes the MPI environment.
`MPI_Comm_rank`	Determines the rank (ID) of the current process.
`MPI_Send`	Sends a message (data) from one process to another.
`MPI_Recv`	Receives a message (data) from another process.
`MPI_Finalize`	Terminates the MPI environment.

Table 1: Key MPI Functions Used

4 Code Structure and Implementation

The implementation uses C with the MPI library for process communication and `libcurl` for HTTP requests.

4.1 Full Source Code

```c
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <curl/curl.h>

#define BUFFER_SIZE 4096

// --- NEW FUNCTION: Callback to safely receive data from Web Server ---
size_t write_callback(void *contents, size_t size, size_t nmemb, void *userp) {
    size_t realsize = size * nmemb;
    char *response_buffer = (char *)userp;

    // Check to avoid buffer overflow
    if (realsize >= BUFFER_SIZE) {
        printf("Warning: Response too large, truncating...\n");
        realsize = BUFFER_SIZE - 1;
    }

    // Copy received data to response variable
    memcpy(response_buffer, contents, realsize);

    // Add null terminator
    response_buffer[realsize] = 0;

    return realsize;
}

```

```
void send_http_request(const char* request, char* response) {
    CURL* curl;
    CURLcode res;

    curl = curl_easy_init();
    if (!curl) {
        fprintf(stderr, "Error initializing libcurl\n");
        strcpy(response, "Error initializing libcurl");
        return;
    }

    // Configure the HTTP request
    // Note: URL from webhook.site for testing
    curl_easy_setopt(curl, CURLOPT_URL, "https://webhook.site/d5ea60ce-050e-47c7-b77c-bd16907f0665");
    curl_easy_setopt(curl, CURLOPT_POSTFIELDS, request);

    // --- UPDATED LOGIC ---
    // Use write_callback instead of direct memcpy for safety
    curl_easy_setopt(curl, CURLOPT_WRITEFUNCTION, write_callback);
    curl_easy_setopt(curl, CURLOPT_WRITEDATA, response);
    // --------------------

    // Perform the HTTP request
    res = curl_easy_perform(curl);
    if (res != CURLE_OK) {
        fprintf(stderr, "curl_easy_perform() failed: %s\n", curl_easy_strerror(res));
        sprintf(response, "HTTP request failed: %s", curl_easy_strerror(res));
    }

    curl_easy_cleanup(curl);
}

void read_file_to_buffer(const char* filename, char* buffer) {
    FILE* file = fopen(filename, "r");
    if (!file) {
        fprintf(stderr, "Error opening file: %s\n", filename);
        return;
    }

    size_t bytesRead = fread(buffer, 1, BUFFER_SIZE - 1, file);
    buffer[bytesRead] = '\0';  // Null-terminate the string
    fclose(file);
}

int main(int argc, char** argv) {
    MPI_Init(&argc, &argv);

    int world_rank, world_size;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

    if (world_size < 2) {
        fprintf(stderr, "World size must be at least 2\n");
        MPI_Abort(MPI_COMM_WORLD, 1);
    }
```

```
84
85      if (world_rank == 0) {
86          // --- CLIENT PROCESS ---
87          char http_request[BUFFER_SIZE] = {0};
88
89          if (argc == 2) {
90              // Read from a file
91              read_file_to_buffer(argv[1], http_request);
92              printf("Client is sending file content: %s\n", http_request
    );
93          } else {
94              // Default message input
95              printf("Enter a message to send: \n");
96              fflush(stdout); // IMPORTANT: Flush stdout to display
    prompt immediately
97
98              fgets(http_request, BUFFER_SIZE, stdin);
99              http_request[strcspn(http_request, "\n")] = 0;  // Remove
    trailing newline
100         }
101
102         MPI_Send(http_request, strlen(http_request) + 1, MPI_CHAR, 1,
    0, MPI_COMM_WORLD);
103
104         // Receive HTTP response from proxy
105         char http_response[BUFFER_SIZE];
106         MPI_Recv(http_response, BUFFER_SIZE, MPI_CHAR, 1, 0,
    MPI_COMM_WORLD, MPI_STATUS_IGNORE);
107         printf("Client received response: %s\n", http_response);
108
109     } else if (world_rank == 1) {
110         // --- PROXY PROCESS ---
111         char http_request[BUFFER_SIZE];
112         MPI_Recv(http_request, BUFFER_SIZE, MPI_CHAR, 0, 0,
    MPI_COMM_WORLD, MPI_STATUS_IGNORE);
113
114         printf("Proxy received request: %s\n", http_request);
115
116         // Forward the request to the HTTP server
117         char http_response[BUFFER_SIZE] = {0};
118         send_http_request(http_request, http_response);
119
120         // Send the response back to the client
121         MPI_Send(http_response, strlen(http_response) + 1, MPI_CHAR, 0,
     0, MPI_COMM_WORLD);
122     }
123
124     MPI_Finalize();
125     return 0;
126 }
```

Listing 1: http_proxy_mpi.c

4.2 Implementation Visuals

Below are the key logic blocks captured from the source code.

```c
if (world_rank == 0) {

    char http_request[BUFFER_SIZE] = {0};

    if (argc == 2) {

        read_file_to_buffer(argv[1], http_request);
        printf("Client is sending file content: %s\n", http_request);
    } else {

        printf("Enter a message to send: \n");
        fflush(stdout);

        fgets(http_request, BUFFER_SIZE, stdin);
        http_request[strcspn(http_request, "\n")] = 0;
    }

    MPI_Send(http_request, strlen(http_request) + 1, MPI_CHAR, 1, 0, MPI_COMM_WORLD);


    char http_response[BUFFER_SIZE];
    MPI_Recv(http_response, BUFFER_SIZE, MPI_CHAR, 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    printf("Client received response: %s\n", http_response);
```

Figure 2: Client Logic (Rank 0): Handling input and sending MPI message.

```c
} else if (world_rank == 1) {

    char http_request[BUFFER_SIZE];
    MPI_Recv(http_request, BUFFER_SIZE, MPI_CHAR, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

    printf("Proxy received request: %s\n", http_request);


    char http_response[BUFFER_SIZE] = {0};
    send_http_request(http_request, http_response);


    MPI_Send(http_response, strlen(http_response) + 1, MPI_CHAR, 0, 0, MPI_COMM_WORLD);
}
```

Figure 3: Proxy Logic (Rank 1): Receiving MPI message, forwarding via HTTP, and returning response.

5 Deployment and Usage Guideline

5.1 Prerequisites

To compile and run this software, the following dependencies are required:

- **GCC Compiler**: Standard C compiler.
- **MPI Implementation**: MPICH or OpenMPI.
- **libcurl**: The multiprotocol file transfer library.

Installation on Kali Linux:

```
sudo apt update
sudo apt install build-essential mpich libcurl4-openssl-dev
```

5.2 Compilation

Use the MPI wrapper compiler `mpicc` and link the curl library:

```
mpicc -o http_proxy_mpi http_proxy_mpi.c -lcurl
```

5.3 Usage Manual

The program supports two modes of operation:

1. Interactive Mode (Default): Run the program without arguments. The client will prompt you to type a message.

```
mpirun -np 2 ./http_proxy_mpi
# Then type your message when prompted
```

2. File Mode: Run the program with a filename argument. The client will read the file content and send it.

```
mpirun -np 2 ./http_proxy_mpi test.txt
```

6 Experimental Results

The system was tested by sending a custom message ”hello world” from the client.

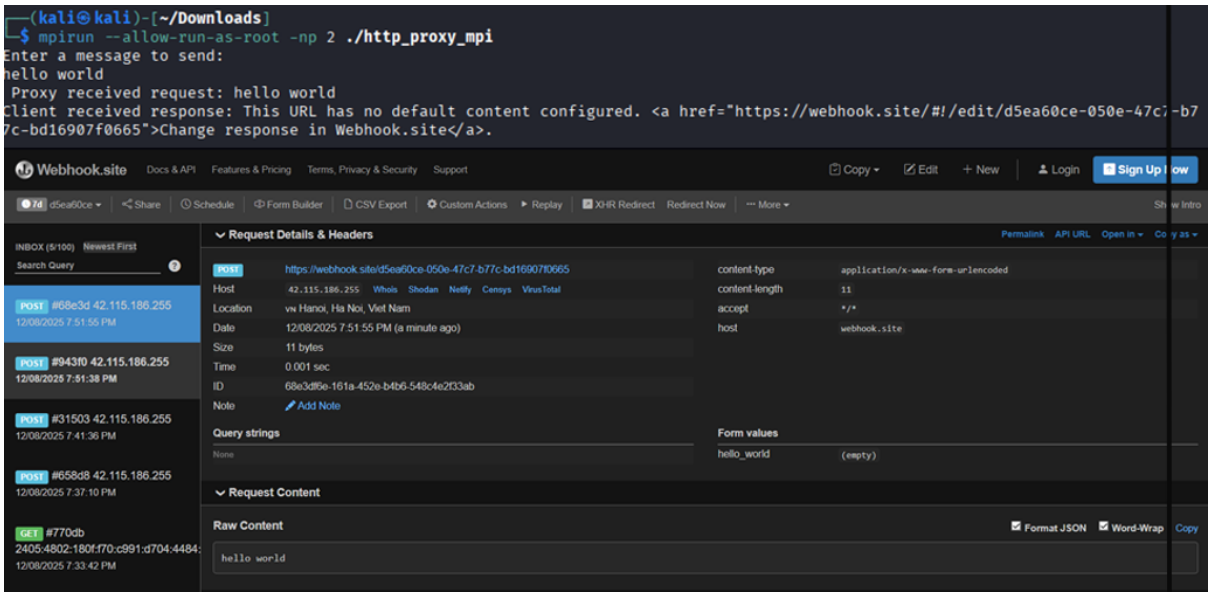

Figure 4: Execution Result: The terminal shows the interaction between Client and Proxy, while the Webhook.site dashboard confirms the receipt of the POST request.

As shown in Figure 4:

1. The Client (Rank 0) prompts for input: ”hello world”.
2. The Proxy (Rank 1) receives the request via MPI.
3. The Webhook.site interface (background) logs the incoming POST request with the content ”hello world”.
4. The Client receives the URL response from the Proxy, completing the cycle.

7 Team Contribution

Member	Contribution
Pham Hoang Viet	System Architect, Core Logic Implementation, Report Writing
Nghiem Xuan Son	Deployment Testing, MPI Configuration, Slide Making
Vu Xuan Thai	Core Logic Implementation, Deployment Testing, Final Editing
Hoang Minh Quan	MPI Configuration, Code Review, Report Writing
Nguyen Ngoc Quang	MPI Configuration, Slide Making, Report Fixing
Pham Huu Minh	Code Review, Testing , Slide Making
Tran Vu Cong Minh	Code Review, Testing, Final Editing

Table 2: Roles and Contributions of Team Members

8 Conclusion

This project successfully demonstrates an HTTP-over-MPI proxy using C. Rank 0 behaves as a client that can send either file contents or user-typed messages, while rank 1 acts as a proxy that forwards the request to an external HTTP service. The design illustrates basic MPI communication combined with network programming, enabling internal nodes to interact with web services securely and efficiently.