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 Overview **3**

3 MPI (Message Passing Interface) Overview **3**
3.1 Key MPI Functions Used 4

4 System Architecture **4**
4.1 Client Process 4
4.2 Proxy Process 5

5 Code Structure and Explanation **5**
5.1 Header Files and Definitions 7
5.2 The `write_callback` and `send_http_request` Functions 7
5.3 The `read_file_to_buffer` Function 7
5.4 The `main` Function 7

6 Flow of Execution **8**
6.1 Client Process (Rank 0) 8
6.2 Proxy Process (Rank 1) 8

7 Error Handling **8**

8 Performance Considerations **8**

9 Conclusion **9**

10 References **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 client can either send a predefined message or the contents of a file. The proxy forwards the request to a specified HTTP server, retrieves the response, and sends it back to the client. This system employs MPI for message passing, making it suitable for parallel computing environments.

2 System Overview

The system involves two key processes:

- **Client Process**: Sends an HTTP request to the proxy, either as a predefined message or the contents of a file.
- **Proxy Process**: Receives the HTTP request from the client, forwards it to an HTTP server (via libcurl), and sends the response back to the client.

The communication between the client and the proxy is managed via MPI. The client sends an HTTP request message, and the proxy forwards it to the HTTP server. The proxy then returns the server's response to the client.

3 MPI (Message Passing Interface) Overview

Primitive	**Meaning**
`MPI_bsend`	Append outgoing message to a local send buffer
`MPI_send`	Send a message and wait until copied to local or remote buffer
`MPI_ssend`	Send a message and wait until receipt starts
`MPI_sendrecv`	Send a message and wait for reply
`MPI_isend`	Pass reference to outgoing message, and continue
`MPI_issend`	Pass reference to outgoing message, and wait until receipt starts
`MPI_recv`	Receive a message; block if there is none
`MPI_irecv`	Check if there is an incoming message, but do not block

Table 1: MPI Primitives and their Meanings

MPI is a standardized and portable message-passing system designed for parallel computing. It enables processes to communicate with each other by sending and receiving messages. This system uses MPI for inter-process communication, specifically:

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

3.1 Key MPI Functions Used

- **MPI_Init**: Initializes the MPI environment.
- **MPI_Comm_rank**: Determines the rank of the process in the communicator.
- **MPI_Comm_size**: Determines the size of the communicator (number of processes).
- **MPI_Send** and **MPI_Recv**: Used for sending and receiving data.

MPI Function	Purpose
`MPI_Init`	Initializes the MPI environment.
`MPI_Comm_rank`	Determines the rank (ID) of the current process within the MPI communicator.
`MPI_Comm_size`	Determines the total number of processes in the MPI communicator.
`MPI_Send`	Sends a message (data) from one process to another.
`MPI_Recv`	Receives a message (data) from another process.
`MPI_Abort`	Aborts all MPI processes in the communicator if an error occurs.
`MPI_Finalize`	Terminates the MPI environment, cleaning up resources used by MPI.

Table 2: Common MPI Functions and their Purposes

4 System Architecture

The system architecture consists of two processes:

- **Client Process (Rank 0)**: Initiates the communication by sending an HTTP request to the proxy. It can either provide the message directly or specify a file to be sent.
- **Proxy Process (Rank 1)**: Acts as an intermediary between the client and the HTTP server. It forwards the client's request to the server and sends back the server's response to the client.

4.1 Client Process

- **Role**: The client process is responsible for sending the HTTP request and receiving the response.
- **Input**: The client can either input a message manually or specify a file from which the content will be sent.
- **Output**: The HTTP response from the server, which is received from the proxy.

4.2 Proxy Process

- **Role**: The proxy process receives the request from the client, forwards it to the HTTP server (using libcurl), and then sends the response back to the client.
- **Input**: The HTTP request from the client.
- **Output**: The HTTP response from the server, which is sent back to the client.

5 Code Structure and Explanation

```
mpicc -o http_proxy_mpi http_proxy_mpi.c -lcurl
mpirun -np 2 ./http_proxy_mpi test.txt
```

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

#define BUFFER_SIZE 4096

size_t write_callback(void *contents, size_t size, size_t nmemb, void *
    userp) {
    size_t realsize = size * nmemb;
    char *response_buffer = (char *)userp;

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

    memcpy(response_buffer, contents, realsize);
    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;
    }

    curl_easy_setopt(curl, CURLOPT_URL,
        "https://webhook.site/d5ea60ce-050e-47c7-b77c-bd16907f0665");
    curl_easy_setopt(curl, CURLOPT_POSTFIELDS, request);
    curl_easy_setopt(curl, CURLOPT_WRITEFUNCTION, write_callback);
    curl_easy_setopt(curl, CURLOPT_WRITEDATA, response);

    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';
    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);
    }

    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);

    } else if (world_rank == 1) {
        char http_request[BUFFER_SIZE];
```

```
99          MPI_Recv(http_request, BUFFER_SIZE, MPI_CHAR,
100                 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
101
102         printf("Proxy received request: %s\n", http_request);
103
104         char http_response[BUFFER_SIZE] = {0};
105         send_http_request(http_request, http_response);
106
107         MPI_Send(http_response, strlen(http_response) + 1,
108                 MPI_CHAR, 0, 0, MPI_COMM_WORLD);
109     }
110
111     MPI_Finalize();
112     return 0;
113 }
```

5.1 Header Files and Definitions

The program includes:

- `mpi.h`: MPI primitives for distributed communication.
- `stdio.h`, `stdlib.h`, `string.h`: standard C I/O and utilities.
- `curl/curl.h`: libcurl API for HTTP requests.

`BUFFER_SIZE` defines the maximum size of request and response buffers. A custom callback `write_callback` is defined to safely receive HTTP response data without overflowing the buffer.

5.2 The `write_callback` and `send_http_request` Functions

`write_callback` is registered as the libcurl write function. It computes the size of the incoming chunk, truncates it if necessary, copies it into the response buffer, and appends a null terminator.

`send_http_request` initializes a CURL handle, sets the webhook URL and POST data, registers `write_callback`, performs the HTTP request, reports errors, and cleans up.

5.3 The `read_file_to_buffer` Function

`read_file_to_buffer` opens a file, reads up to `BUFFER_SIZE - 1` bytes, null-terminates the result, and closes the file. It is used when the client is launched with a filename as command-line argument.

5.4 The `main` Function

The `main` function initializes MPI, checks that at least two processes exist, and then runs the client on rank 0 and the proxy on rank 1.

6 Flow of Execution

6.1 Client Process (Rank 0)

- If a filename is given (`argc == 2`), the client loads the file content into `http_request`; otherwise it prompts the user to enter a message and flushes `stdout` so the prompt is visible immediately.
- The client sends the request to rank 1 with `MPI_Send`.
- The client waits for the response from rank 1 with `MPI_Recv` and prints it.

6.2 Proxy Process (Rank 1)

- The proxy receives the request from rank 0.
- It calls `send_http_request`, which forwards the payload to the configured webhook URL and stores the HTTP response in `http_response`.
- The proxy sends the response back to the client via `MPI_Send`.

7 Error Handling

Error handling occurs at several levels:

- If `world_size < 2`, the program prints an error and calls `MPI_Abort`.
- If a file cannot be opened in `read_file_to_buffer`, an error message is printed to `stderr`.
- If libcurl cannot be initialized, `send_http_request` writes a descriptive error into the response buffer.
- If `curl_easy_perform` fails, the libcurl error string is printed and also written into the response buffer as `"HTTP request failed: ..."`.
- `write_callback` guards against buffer overflow by truncating excessively large responses.

8 Performance Considerations

The system uses two MPI processes and fixed-size buffers, so MPI overhead is minimal and memory management is straightforward. The dominant performance factor is the HTTP round-trip time to the webhook server. The additional safety checks in `write_callback` add negligible overhead compared with network latency.

9 Conclusion

This project implements an HTTP-over-MPI proxy using C, MPI, and libcurl. 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 and relays the response. The design illustrates basic MPI communication, integration with a network API, and safe handling of variable-length responses.

10 References

- https://github.com/mpitutorial/mpitutorial/tree/gh-pages/tutorials/
- https://mpitutorial.com/tutorials/
- https://curc.readthedocs.io/en/latest/programming/MPI-C.html