University of Science and Technology of Hanoi

Distributed System

Midterm report

HTTP over MPI

Group	:	*09*
Members	:	*Nguyen Tai Anh - 22BI13028*
	:	*Do Bao Nhi - 22BI13349*
	:	*Do Trong Dat - 22BI13075*
	:	*Hoang Quan - 22BI13368*
	:	*Le Van Truong - 22BI13440*
	:	*Nguyen Hai Nam - 22BI13322*
Major	:	*Cyber Security*
Lecturer	:	*MSc. Le Nhu Chu Hiep*

Hanoi, December 31 2024

Contents

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

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

#define BUFFER_SIZE 4096

void send_http_request(const char* request, char* response) {
    CURL* curl;
    CURLcode res;
    struct curl_slist* headers = NULL;

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

    // Configure the HTTP request
    curl_easy_setopt(curl, CURLOPT_URL, "https://webhook.site/e5022f2e-5e9a-4290-9f06-
    cb1c906a5d78");
    curl_easy_setopt(curl, CURLOPT_POSTFIELDS, request);

    // Set up a response handler
    curl_easy_setopt(curl, CURLOPT_WRITEFUNCTION, memcpy);
    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));
        strcpy(response, "HTTP request failed");
    }

    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);
    }

    if (world_rank == 0) {
        // Client process: Decide whether to send a custom message or a file
        char http_request[BUFFER_SIZE] = {0};

        if (argc == 2) {
            // Read from a file
            read_file_to_buffer(argv[1], http_request);
            printf("Client is sending file content: %s\n", http_request);
        } else {
            // Default message input
            printf("Enter a message to send: ");
            fgets(http_request, BUFFER_SIZE, stdin);
            http_request[strcspn(http_request, "\n")] = 0;  // Remove trailing newline
        }

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

        // Receive HTTP response from proxy
        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) {
        // Proxy process: Receive HTTP request from client
        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);

        // Forward the request to the HTTP server
        char http_response[BUFFER_SIZE] = {0};
        send_http_request(http_request, http_response);

        // Send the response back to the client
        MPI_Send(http_response, strlen(http_response) + 1, MPI_CHAR, 0, 0,
    MPI_COMM_WORLD);
    }

    MPI_Finalize();
    return 0;
}
```

5.1 Header Files and Definitions

The code begins by including the necessary libraries:

- `mpi.h`: Provides the MPI functionalities.
- `stdio.h`: Standard input/output for printing logs and error messages.
- `stdlib.h`: Standard library for memory allocation and other utilities.
- `string.h`: For string manipulation.
- `curl/curl.h`: The libcurl library for making HTTP requests.

A constant `BUFFER_SIZE` is defined to allocate memory for the HTTP request and response.

5.2 The `send_http_request` Function

This function is responsible for sending an HTTP POST request to the HTTP server using libcurl. It performs the following:

- Initializes a libcurl handle using `curl_easy_init()`.
- Sets the URL and request data using `curl_easy_setopt()`.
- Defines the write function to handle the response using `curl_easy_setopt(CURLOPT_WRITEFUNCTI`
- Performs the HTTP request with `curl_easy_perform()`.
- Handles errors and cleans up resources with `curl_easy_cleanup()`.

5.3 The `read_file_to_buffer` Function

This function reads the contents of a file into a buffer:

- It opens the file in read mode.
- Reads the content into a buffer, ensuring that the size doesn't exceed `BUFFER_SIZE`.
- Closes the file after reading.

5.4 The `main` Function

The `main` function initializes MPI, checks the number of processes, and assigns roles to the client and proxy processes:

- **Client Process** (rank 0): If a filename is provided as a command-line argument, the client reads the file and sends it. Otherwise, the client prompts the user for a message and sends it. The client then waits for a response from the proxy.
- **Proxy Process** (rank 1): The proxy receives the HTTP request from the client, forwards it to the HTTP server using `send_http_request()`, and sends the response back to the client.

6 Flow of Execution

6.1 Client Process

- The client process determines if a file is specified. If not, it prompts the user for a message.
- The client sends the HTTP request to the proxy via MPI.
- The client waits for and receives the HTTP response from the proxy.

6.2 Proxy Process

- The proxy receives the HTTP request from the client via MPI.
- The proxy forwards the request to the HTTP server using libcurl.
- The proxy receives the server's response and sends it back to the client.

7 Error Handling

Error handling is done at key points in the program:

- If the `curl_easy_init()` fails, the function prints an error and returns a message to the client.
- If there is an issue performing the HTTP request, the function logs the error using `curl_easy_strerror()`.
- The program ensures that the number of processes is at least 2, otherwise, it aborts with an error message.

8 Performance Considerations

The performance of this system depends on the network latency between the client, proxy, and HTTP server. MPI's communication overhead could be a bottleneck if the number of processes increases significantly, but it is suitable for scenarios with a small number of processes (in this case, two).

9 Conclusion

This MPI-based HTTP request system provides a simple yet effective way to send HTTP requests from a client to an HTTP server via a proxy. It demonstrates the use of MPI for inter-process communication and libcurl for HTTP interactions. The ability to send either custom messages or file contents adds flexibility to the system.

10 References

- MPI Documentation: The official MPI documentation for understanding the MPI standard and functions.
- libcurl Documentation: The libcurl documentation for detailed usage of the HTTP request functions.
- https://github.com/mpitutorial/mpitutorial/tree/gh-pages/tutorials/