MPI-based Multi-Client Remote Shell Evolution from v1 to v5

Nguyen Tien Dung – 23BI14116

December 4, 2025

Abstract

This report describes the design and implementation of a multi-client remote shell application built on top of MPI. The project was developed incrementally through five versions (v1–v5). Each version introduces new features and architectural improvements, from a basic single-server prototype to a dispatcher–worker architecture with a worker pool, history, logging, basic security policies, toy encryption and both scripted and interactive clients.

The first part of the report summarizes the evolution from v1 to v5 and highlights the main changes. The second part focuses in detail on the final version (v5), including its architecture, communication protocol, implementation structure, benchmarking method and possible future extensions.

1 Introduction

The goal of this midterm project is to design and implement a *remote shell* service using MPI (Message Passing Interface). Instead of using TCP sockets or SSH, the system relies entirely on MPI message passing.

The requirements are:

- Support **multiple clients** concurrently.
- Execute arbitrary shell commands on a remote process.
- Send the command output back to the client.
- Use MPI primitives (`MPI_Send`, `MPI_Recv`, etc.) for all communication.

In addition to these minimal requirements, I also explored several advanced features typically relevant in distributed systems: simple security policies, basic encryption, logging, history, a worker pool and load balancing.

2 System overview

At a high level, the system is an MPI application launched as:

```
mpirun -np N ./mpi_remote_shell_v5 [interactive]
```

The MPI ranks are divided into three roles:

- **Rank 0 – Dispatcher**: receives commands from clients, routes them to worker processes, and sends back the results.
- **Ranks 1..W – Workers**: execute commands on behalf of clients (remote shell servers).
- **Ranks** $> W$ **– Clients**: send commands and receive outputs. Each client can be scripted or interactive.

The design evolved gradually from a simple single-server prototype. The next section summarizes the evolution from v1 to v5.

3 Evolution from v1 to v5

3.1 Version 1: single server, scripted clients

In the first version, the system consisted of one server (rank 0) and one or more scripted clients (rank ≥ 1):

- Each client had a hard-coded list of commands in an array.
- The client loop was: send command string to rank 0, receive the result, print it.
- The server used `MPI_ANY_SOURCE` to receive commands from any client, executed the command using `popen()`, and sent back the output.
- `"exit"` or `"quit"` from a client indicated that the client was done.

This version satisfied the basic "remote shell using MPI (multiple clients)" requirement, but had several limitations:

- No logging or command history.
- No security controls.
- No interactive client.
- No separation between routing and execution.

3.2 Version 2: logging and control commands

Version 2 extended the single-server design with extra management features:

- **Logging**: the server appended all received commands to a log file (`server_log.txt`) with timestamps.
- **History**: the server maintained a ring buffer of recent commands and could return it on request.
- **Control commands**:
  - `__clients`: list active client ranks.
  - `__history`: show recent commands.
  - `__serverinfo`: run `uname -a` and `hostname`.
- Clients were still scripted but now used these special commands as well as normal shell commands.

This version introduced the idea of a higher-level protocol on top of MPI (distinguishing between normal and control commands), but the architecture was still a single server.

3.3 Version 3: interactive client and remote file read

Version 3 added two user-facing features:

- **Interactive client mode**: by running `mpirun -np N ./prog interactive`, each client could read commands from `stdin` in a loop, send them to the server, and display the output. This made the system feel closer to a minimal SSH session.
- **Remote file read** via `__get <path>`: the server opened the requested file, read its contents, and sent them back to the client (up to a fixed buffer size with truncation notice).

The architecture was still single server, but the application became more practical and interactive.

3.4 Version 4: toy encryption, security policy, script files

Version 4 focused on security, usability and experimentation support:

- **Toy encryption**: a simple XOR-based scheme was added between clients and the server. All command and result payloads were XORed with a one-byte key before being sent, and XORed again on reception.
  - This is *not* real cryptography, but it shows that application-level transformations can be layered on top of MPI.
- **Security policy**: basic blocking of dangerous commands, e.g. `rm -rf`, a fork bomb function, `mkfs`, `dd` targeting `/dev` devices.
- **Script files**: instead of always hard-coding client scripts, each client could optionally read commands from a file `script_<rank>.txt`.
- **CSV logging**: in addition to human-readable logs, commands were also logged in a CSV file for later analysis or plotting.

This version made the prototype more realistic from a security and experiment perspective, but the underlying architecture was still one server and multiple clients.

3.5 Version 5: dispatcher–worker architecture (final)

The final version (v5) introduces a more distributed architecture with three roles:

- Rank 0 is a **dispatcher**.
- Ranks 1..W are **worker servers**.
- Ranks $> W$ are **clients**.

Key properties:

- Clients never directly send commands to the workers. They only talk to the dispatcher.
- The dispatcher performs **routing and load balancing**, forwarding commands to workers in a simple round-robin fashion.
- Workers are dedicated to **command execution**, including policies, `__help`, `__get`, `__history` and shell execution via `popen()`.
- When all clients have disconnected, the dispatcher sends a shutdown message to all workers and terminates.

The following table summarizes the evolution:

Version	Architecture	Main features
v1	Single server	Scripted clients, basic remote shell via MPI.
v2	Single server	Logging, history, control commands.
v3	Single server	Interactive client mode, remote file read.
v4	Single server	Toy XOR encryption, security policy, scripts from file, CSV logs, benchmark.
v5	Dispatcher + workers	Worker pool, load balancing, clients talk only to dispatcher; all v4 features reused in workers/clients.

The rest of this report focuses on the final version.

4 Detailed design of Version 5

4.1 Process roles and topology

Let N be the total number of MPI processes. Version 5 uses the following mapping:

- Rank 0: dispatcher.
- Ranks $1..W$: worker processes, where $W \geq 1$ and $W \leq N - 2$.
- Ranks $> W$: client processes.

The program automatically chooses the number of workers based on N, ensuring there is at least one worker and one client. For example:

- With `mpirun -np 4`, the system uses 1 dispatcher, 1 worker, and 2 clients.
- With `mpirun -np 5`, it uses 1 dispatcher, 2 workers and 2 clients.

4.2 Communication protocol

The protocol introduces four MPI tags to distinguish message flows:

- `TAG_CMD_CLIENT` (0): client → dispatcher (command).
- `TAG_RESULT_CLIENT` (1): dispatcher → client (result).
- `TAG_CMD_WORKER` (2): dispatcher → worker (job assignment).
- `TAG_RESULT_WORKER` (3): worker → dispatcher (execution result).

For messages between dispatcher and workers, the following structs are used:

Listing 1: Structs for dispatcher–worker communication

```
typedef struct {
    int client_rank;
    char cmd[MAX_CMD];
} WorkerRequest;

typedef struct {
    int client_rank;
    char result[MAX_OUTPUT];
} WorkerResponse;
```

These are sent using:

```
MPI_Send(&req, sizeof(req), MPI_BYTE, worker_rank, TAG_CMD_WORKER, MPI_COMM_WORLD);
MPI_Recv(&resp, sizeof(resp), MPI_BYTE, worker_rank, TAG_RESULT_WORKER, MPI_COMM_WORLD
    , &status);
```

For client–dispatcher messages, payloads are simple character buffers (`char cmd_buf[MAX_CMD]` and `char result_buf[MAX_OUTPUT]`), optionally wrapped by an XOR operation for toy encryption.

4.3 Client behavior

Clients can operate in two modes:

- **Scripted mode** (default): each client either reads commands from `script_<rank>.txt` or falls back to a built-in script. After sending each command to the dispatcher, the client waits for a result and prints it.

- **Interactive mode**: if the program is started with the argument `interactive`, each client enters a loop reading commands from `stdin`, sending them to the dispatcher, and printing results until the user types `exit` or `quit`.

In both modes, the client uses helper functions to send command strings and receive results with XOR applied:

Listing 2: Client side: encrypted send/receive

```
static void send_encrypted_cmd_from_client(char *cmd_buf, int dest_rank) {
    xor_buffer(cmd_buf, MAX_CMD, ENC_KEY);
    MPI_Send(cmd_buf, MAX_CMD, MPI_CHAR,
            dest_rank, TAG_CMD_CLIENT, MPI_COMM_WORLD);
    xor_buffer(cmd_buf, MAX_CMD, ENC_KEY); // restore local copy
}

static void recv_encrypted_result_at_client(char *res_buf, int src_rank) {
    MPI_Status status;
    MPI_Recv(res_buf, MAX_OUTPUT, MPI_CHAR,
            src_rank, TAG_RESULT_CLIENT, MPI_COMM_WORLD, &status);
    xor_buffer(res_buf, MAX_OUTPUT, ENC_KEY);
}
```

Clients send the string `"exit"` to close their session. The dispatcher interprets this as a disconnection request and eventually terminates the whole system when all clients have left.

4.4 Dispatcher behavior (rank 0)

The dispatcher is the central coordination point. Its responsibilities are:

- Receive commands from any client using `MPI_ANY_SOURCE`.
- Maintain a set of active client ranks.
- Handle some control commands locally (e.g. `__clients`).
- Forward all other commands to workers.
- Forward worker results back to the correct client.
- Shut down the workers when no clients remain.

The dispatcher loop receives commands from clients, decrypts them, processes special commands and forwards jobs to workers via `WorkerRequest`:

Listing 3: Dispatcher main loop (simplified)

```
static void dispatcher_loop(int world_size, int num_workers) {
    int active_clients[MAX_CLIENTS] = {0};
    int total_clients = 0;

    // clients are ranks (num_workers+1)..(world_size-1)
    for (int r = num_workers + 1; r < world_size && r < MAX_CLIENTS; r++) {
        active_clients[r] = 1;
        total_clients++;
    }

    int next_worker_index = 0;
    char cmd_buf[MAX_CMD];
    char result_buf[MAX_OUTPUT];

    while (total_clients > 0) {
        MPI_Status status;
```

```
        recv_encrypted_cmd_at_dispatcher(cmd_buf, &status);
        int client_rank = status.MPI_SOURCE;
        cmd_buf[MAX_CMD - 1] = '\0';

        // client exit
        if (strcmp(cmd_buf, "exit") == 0 || strcmp(cmd_buf, "quit") == 0) {
            snprintf(result_buf, sizeof(result_buf),
                    "Client %d disconnected.\n", client_rank);
            send_encrypted_result_from_dispatcher(result_buf, client_rank);
            if (active_clients[client_rank]) {
                active_clients[client_rank] = 0;
                total_clients--;
            }
            continue;
        }

        // handled locally: __clients
        if (strcmp(cmd_buf, "__clients") == 0) {
            int len = snprintf(result_buf, sizeof(result_buf),
                               "Active clients: ");
            for (int r = num_workers + 1; r < world_size; r++) {
                if (active_clients[r]) {
                    len += snprintf(result_buf + len,
                                    sizeof(result_buf) - len,
                                    "%d ", r);
                }
            }
            snprintf(result_buf + len,
                    sizeof(result_buf) - len, "\n");
            send_encrypted_result_from_dispatcher(result_buf, client_rank);
            continue;
        }

        // forward to worker
        int worker_rank = 1 + (next_worker_index % num_workers);
        next_worker_index++;

        WorkerRequest req;
        req.client_rank = client_rank;
        strncpy(req.cmd, cmd_buf, MAX_CMD - 1);
        req.cmd[MAX_CMD - 1] = '\0';

        MPI_Send(&req, sizeof(req), MPI_BYTE,
                worker_rank, TAG_CMD_WORKER, MPI_COMM_WORLD);

        WorkerResponse resp;
        MPI_Status wstatus;
        MPI_Recv(&resp, sizeof(resp), MPI_BYTE,
                worker_rank, TAG_RESULT_WORKER, MPI_COMM_WORLD, &wstatus);

        strncpy(result_buf, resp.result, MAX_OUTPUT - 1);
        result_buf[MAX_OUTPUT - 1] = '\0';

        send_encrypted_result_from_dispatcher(result_buf, resp.client_rank);
    }

    // shut down workers
    for (int w = 1; w <= num_workers; w++) {
```

```
        WorkerRequest req;
        req.client_rank = -1;
        strncpy(req.cmd, "__shutdown_worker", MAX_CMD - 1);
        req.cmd[MAX_CMD - 1] = '\0';
        MPI_Send(&req, sizeof(req), MPI_BYTE,
                 w, TAG_CMD_WORKER, MPI_COMM_WORLD);
    }
}
```

4.5 Worker behavior

Workers are responsible for executing commands. They receive `WorkerRequest` messages from the dispatcher, run the appropriate operation, and send back `WorkerResponse` messages.

The worker loop:

- Maintains a local command history (per worker).
- Logs commands with timestamps.
- Checks the security policy to block dangerous commands.
- Handles special commands:
  - `__history`: return the worker's history.
  - `__serverinfo`: return kernel and hostname.
  - `__get <path>`: read a file on the worker node.
  - `__help`: show available special commands.
- For ordinary commands, calls `popen()` and returns the command output.

A simplified worker loop:

Listing 4: Worker loop (simplified)

```
static void worker_loop(int rank) {
    char history[HISTORY_SIZE][MAX_CMD];
    int hist_count = 0;

    while (1) {
        WorkerRequest req;
        MPI_Status status;
        MPI_Recv(&req, sizeof(req), MPI_BYTE,
                 0, TAG_CMD_WORKER, MPI_COMM_WORLD, &status);

        if (req.client_rank == -1 &&
            strcmp(req.cmd, "__shutdown_worker") == 0) {
            break; // exit worker
        }

        const char *cmd = req.cmd;
        int client_rank = req.client_rank;
        char result_buf[MAX_OUTPUT];

        if (is_blocked_command(cmd)) {
            snprintf(result_buf, sizeof(result_buf),
                     "Command blocked by security policy.\n");
        } else if (strcmp(cmd, "__history") == 0) {
            // build history string ...
        } else if (strcmp(cmd, "__serverinfo") == 0) {
            run_command("uname -a; echo; hostname", result_buf,
                        sizeof(result_buf));
```

```
        } else if (strncmp(cmd, "__get ", 6) == 0) {
            const char *path = cmd + 6;
            // read_file_content(path, result_buf, sizeof(result_buf));
        } else if (strcmp(cmd, "__help") == 0) {
            snprintf(result_buf, sizeof(result_buf),
                    "Available special commands:\n"
                    " __help, __history, __serverinfo, __get <path>, ...\n");
        } else {
            run_command(cmd, result_buf, sizeof(result_buf));
        }

        WorkerResponse resp;
        resp.client_rank = client_rank;
        strncpy(resp.result, result_buf, MAX_OUTPUT - 1);
        resp.result[MAX_OUTPUT - 1] = '\0';

        MPI_Send(&resp, sizeof(resp), MPI_BYTE,
                0, TAG_RESULT_WORKER, MPI_COMM_WORLD);
    }
}
```

4.6 Security and toy encryption

The system uses a simple XOR-based transformation between clients and the dispatcher. This is intentionally not strong cryptography—it is only meant to demonstrate that application-level processing (e.g. obfuscation, compression, encryption) can be layered on top of MPI.

The security policy is implemented as a small function that rejects some high-risk commands:

Listing 5: Simple security policy

```
static int is_blocked_command(const char *cmd) {
    if (strstr(cmd, "rm -rf") != NULL) return 1;
    if (strstr(cmd, ":(){:|:&};:") != NULL) return 1;
    if (strncmp(cmd, "mkfs", 4) == 0) return 1;
    if (strncmp(cmd, "dd ", 3) == 0 && strstr(cmd, " /dev/") != NULL) return 1;
    return 0;
}
```

In a real deployment, stronger mechanisms (authentication, encryption, sandboxing) would be required. The project highlights these considerations in the discussion.

5 Benchmarking and evaluation

To evaluate the system, a simple throughput benchmark is implemented in the client. The benchmark client sends a sequence of `echo` commands and measures the round-trip time using `MPI_Wtime()`:

Listing 6: Benchmark client

```
static void benchmark_client(int rank, int dispatcher_rank, int iterations) {
    char cmd_buf[MAX_CMD];
    char result_buf[MAX_OUTPUT];

    double t0 = MPI_Wtime();
    for (int i = 0; i < iterations; i++) {
        snprintf(cmd_buf, sizeof(cmd_buf),
                "echo bench_%d_from_client_%d", i, rank);
```

```
        send_encrypted_cmd_from_client(cmd_buf, dispatcher_rank);
        recv_encrypted_result_at_client(result_buf, dispatcher_rank);
    }
    double t1 = MPI_Wtime();
    double elapsed = t1 - t0;
    double cps = (elapsed > 0.0) ? (iterations / elapsed) : 0.0;

    printf("[Client %d] Benchmark: time = %.4f s, "
           "throughput = %.2f cmd/s\n",
           rank, elapsed, cps);
}
```

An example configuration for experiments:

- **Case 1**: 1 dispatcher, 1 worker, 1 benchmark client, 1 normal client (`-np 4`).
- **Case 2**: 1 dispatcher, 2 workers, 1 benchmark client, 1 normal client (`-np 5`).

For each case, the benchmark is run with a fixed number of commands (e.g. 50 or 100). The output shows the total time and the number of commands per second. Commands are also logged in CSV format by the dispatcher, making it easy to import the data into tools such as Python or Excel for further analysis.

6 Discussion and future work

The final version (v5) demonstrates several important patterns in distributed systems:

- Decoupling of **routing** (dispatcher) and **execution** (workers).
- Use of a **worker pool** to process remote commands.
- Multi-client communication over MPI using message tags and structured payloads.

Several extensions could be implemented in future work:

- **Non-blocking communication:** the dispatcher currently uses blocking receives. Using `MPI_Irecv` and `MPI_Test` could allow overlapping communication and more sophisticated scheduling.
- **Dynamic worker management:** adding or removing workers at runtime, or detecting worker failures and rescheduling commands.
- **Stronger security:** replacing the toy XOR encryption with real cryptography, authenticating clients, and sandboxing command execution.
- **Richer protocol:** supporting persistent sessions, environment variables, or multiple concurrent shell channels per client.

Even without these extensions, the current implementation is sufficient to show how MPI can be used to build a non-trivial distributed application beyond classical numerical computing.

7 Personal work

For this project I:

- Designed the overall architecture for a multi-client remote shell on top of MPI.
- Implemented five versions (v1–v5), gradually adding features such as logging, history, control commands, interactive mode, toy encryption, security policies, script files and a dispatcher–worker architecture.

- Wrote and tested the C code for the final version (`mpi_remote_shell_v5.c`), including the dispatcher loop, worker loop and client logic.
- Designed and ran simple throughput benchmarks and logging to validate the system behavior.
- Prepared this report summarizing the evolution and detailing the final design.

8 Conclusion

This midterm project demonstrates how MPI can be used to implement a distributed remote shell service with multiple clients, a worker pool and basic management and security features. By evolving the design from a simple single-server prototype to a dispatcher–worker architecture, I was able to explore several important concepts in distributed systems, including message-based protocols, load balancing, logging, policy enforcement and evaluation.

The final version (v5) provides a solid base for further experimentation with fault tolerance, non-blocking communication and stronger security, while remaining relatively compact and easy to understand.