University Of Science And Technology Of Hanoi

Practical 3

MPI File Transfer System

Luong Quynh Nhi, 23BI14356, Cyber Security

1. Introduction

This project implements a distributed file-transfer system using the Message Passing Interface (MPI) model.
Instead of building a traditional client–server architecture with plain TCP sockets only, this design integrates MPI workers that cooperate as a worker pool, coordinated by a single master process (rank 0).

This design enables:

- Parallel file processing
- Worker-to-worker communication
- Scalable distributed transfers
- Background message passing between workers

The system is implemented using mpi4py, a Python wrapper over an MPI implementation (MPICH/MS-MPI).

2. Why I Chose This MPI Implementation

Reason 1: mpi4py is the only practical MPI library for Python

- mpi4py provides:
    + Python bindings to any MPI implementation (MPICH, OpenMPI, or Microsoft MPI)
    + Simple send/recv APIs matching the MPI standard
    + Support for tags, ranks, non-blocking probe, collective broadcast

Reason 2: Works on Windows

- Windows does not support OpenMPI.
- The only reliable options are: Microsoft MPI (MS-MPI), MPICH (Windows build), mpi4py supports both without code changes.

Reason 3: Allows worker pool parallelism

- MPI makes it easy to spawn multiple workers: mpiexec -n 5 python server.py

- Rank 0 becomes the master; ranks 1…N become workers.

3. MPI Service Design

3.1 MPI Service Architecture

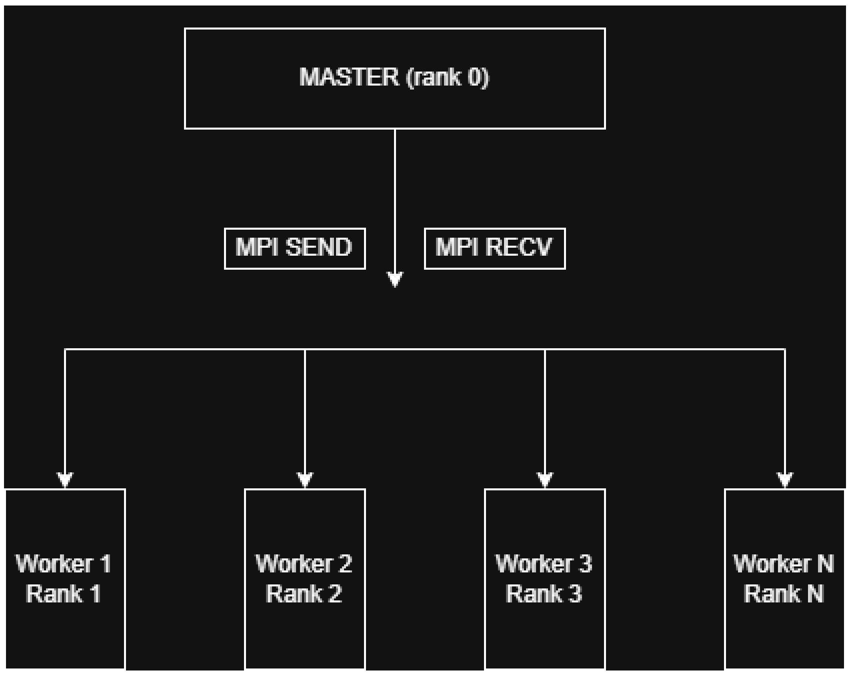
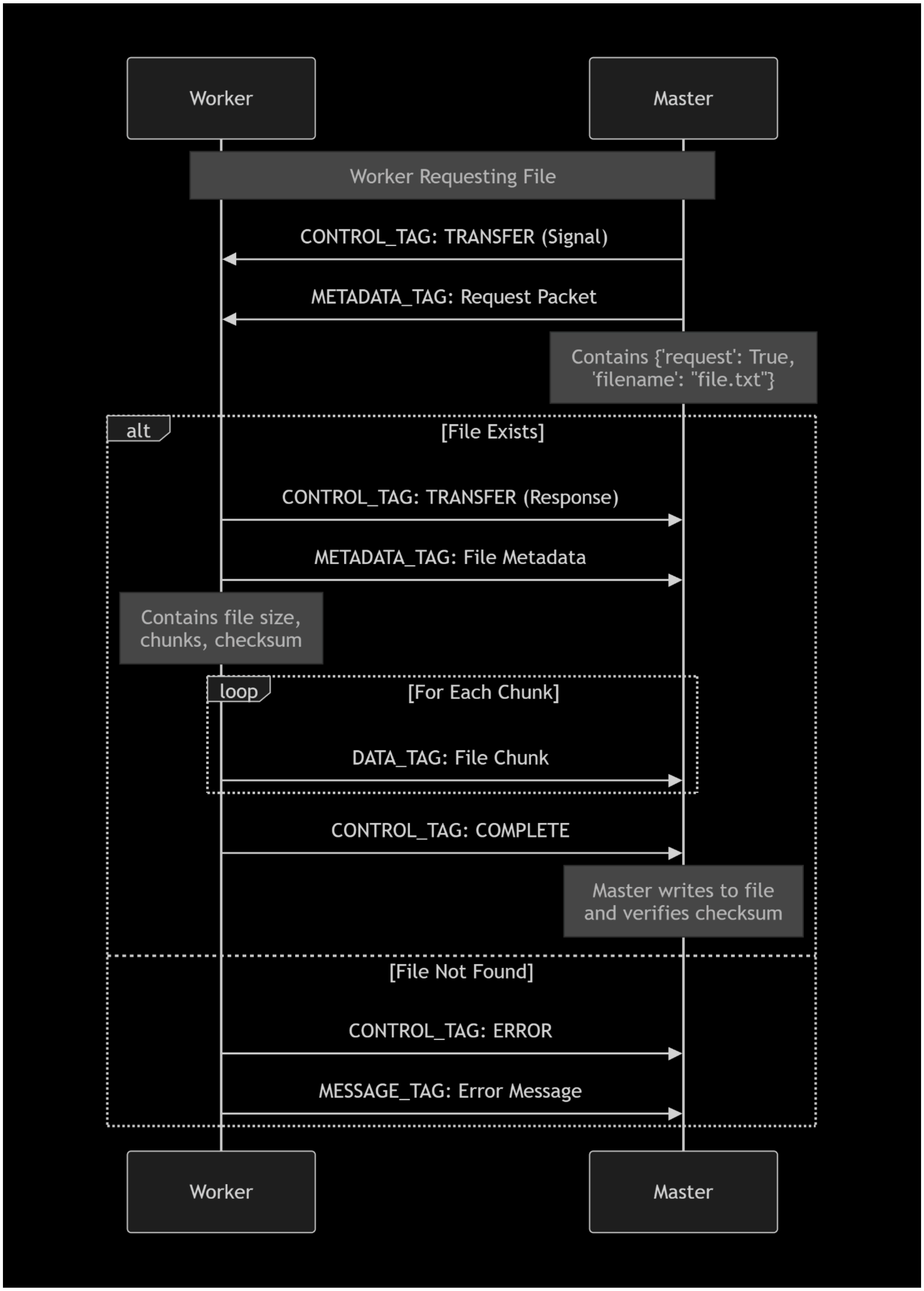


Figure 1 — MPI Master and Worker Roles

Workers also communicate between each other using MPI send/recv.

3.2 Sequence Diagram


Worker
Master
Worker Requesting File
CONTROL_TAG: TRANSFER (Signal)
METADATA_TAG: Request Packet
Contains {'request': True, 'filename': "file.txt"}
alt
[File Exists]
CONTROL_TAG: TRANSFER (Response)
METADATA_TAG: File Metadata
Contains file size, chunks, checksum
loop
[For Each Chunk]
DATA_TAG: File Chunk
CONTROL_TAG: COMPLETE
Master writes to file and verifies checksum
[File Not Found]
CONTROL_TAG: ERROR
MESSAGE_TAG: Error Message
Worker
Master

The sequence diagram shows how a Worker process requests a file from the Master in an MPI-based file-transfer system. The Worker first sends a control signal and a metadata packet containing the filename. The Master checks whether the file exists.

If the file is available, the Master replies with file metadata (size, number of chunks) and then sends the file in multiple data chunks inside a loop. After finishing the transfer, the Master sends a completion signal so the Worker can finalize the file.

If the file does not exist, the Master instead returns an error signal and an error message.
Overall, the sequence illustrates a simple, message-driven protocol using MPI tags to separate control, metadata, and file-data communication.

4. Execution and Result

Scenario: File sending of Master to Workers
Code Execution (Master Side):

```
def master_send(self, filepath, worker_rank):
    """Master sends file to worker"""
    if not 1 <= worker_rank < self.size:
        print(f"Error: Invalid worker rank")
        return

    if not os.path.exists(filepath):
        print(f"Error: File '{filepath}' not found")
        return

    info = self.get_file_info(filepath)
    print(f"\n[Master] Sending '{info['name']}' to Worker {worker_rank}")

    self.comm.send(TRANSFER, dest=worker_rank, tag=CONTROL_TAG)
    self.comm.send({'from': 0, 'info': info}, dest=worker_rank,
tag=METADATA_TAG)

   with open(filepath, 'rb') as f:
      for i in range(info['chunks']):
         chunk_size = info['last'] if i == info['chunks']-1 else CHUNK_SIZE
         self.comm.send(f.read(chunk_size), dest=worker_rank,
tag=DATA_TAG)
         if (i+1) % 5 == 0:
            print(f"  Progress: {i+1}/{info['chunks']} chunks")

   self.comm.send(COMPLETE, dest=worker_rank, tag=CONTROL_TAG)
   print(f"[Master] Transfer complete!")
```

Code Execution (Worker Side):

```
def worker_receive_from_master(self):
   """Worker receives file from master"""
   try:
      data = self.comm.recv(source=0, tag=METADATA_TAG)

      if data.get('request'):
         filename = data['filename']
         if os.path.exists(filename):
            self.worker_log(f"Master requested file: {filename}")
            self.worker_send_to_master(filename)
         else:
            self.worker_log(f"Error: File '{filename}' not found")
         return

      info = data['info']
      self.worker_log(f"Receiving '{info['name']}' from Master")

      filename = f"from_master_{info['name']}"
      total = 0

      with open(filename, 'wb') as f:
         while True:
            if self.comm.Iprobe(source=0, tag=CONTROL_TAG):
```

```
            sig = self.comm.recv(source=0, tag=CONTROL_TAG)
            if sig == COMPLETE:
                break

        if self.comm.Iprobe(source=0, tag=DATA_TAG):
            chunk = self.comm.recv(source=0, tag=DATA_TAG)
            f.write(chunk)
            total += len(chunk)
            progress = (total / info['size']) * 100
            if int(progress) % 20 == 0:
                self.worker_log(f"  Progress: {progress:.0f}%")

    # Verify
    if os.path.getsize(filename) == info['size']:
        if self.checksum(filename) == info['checksum']:
            self.worker_log(f"Saved: {filename} ")
        else:
            self.worker_log(f"Saved: {filename} (checksum mismatch)")
    else:
        self.worker_log("Size mismatch!")

except Exception as e:
    self.worker_log(f"Receive error: {e}")
```

Result:

```
master> send worker_1.log 2

[Master] Sending 'worker_1.log' to Worker 2
[Master] Transfer complete!
```

5. Conclusion

This project demonstrates how MPI can be used not only for scientific computing but also for structured communication systems like file transfer.

By separating messages using MPI tags and implementing a clean Master–Worker protocol, the system allows multiple clients to request files concurrently from a single master. MS-MPI was chosen for its stability on Windows and compatibility with Python.

The final result is a simple but functional MPI-based distributed file transfer system.