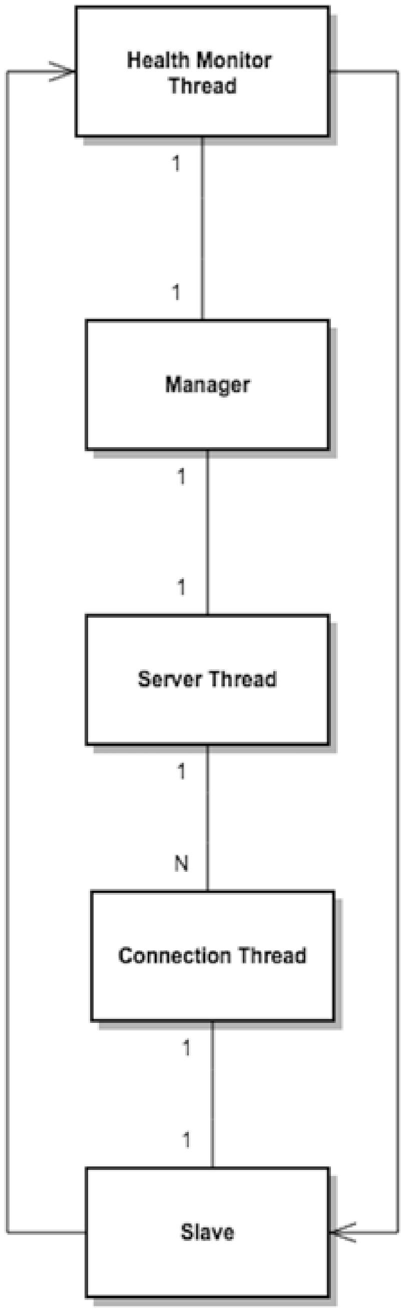
15-440/15-640 Project 1 Report
Process Migration Framework

Team : Lixun Mao (lmao), Druhin Sagar Goel (dsgoel)

1. Framework & Design

1.1. Architecture

1.1.1. Design Outline



Our framework is based on a combination of the Master-Slave model and Peer-To-Peer model. We have the following major classes in our design:

Manager: The Manager class is responsible for managing the whole framework. It is an executable and when run, it begins a console (similar to a command prompt), and waits for the user to type in commands, telling it what to do. At the same time, this class initiates a Server thread which starts listening for Slaves to connect to the framework. Each time a Slave connects to the Server, the user is notified on the console of the newly established connection, along with a Slave ID (unique identifier for the slave). The Manager class also intializes a timer, which after a certain period, sends a health check message to each Slave to keep track of the status of the Slaves in the framework. Interacting with the console has been explained further in Build & Run and Test sections below.

Server: The Server class is responsible for maintaining connections with the Slaves in the framework. When run, it starts listening for incoming Slave connections. Each time a Slave connects to the Server, the Server intiates a Connection class in a separate thread. Upon creating the Connection, the Server notifies the Manager of this new Connection and passes it to the Manager which mainains a map from Slave IDs to Connection.

Connection: There is one Connection class per Slave in the system. Each Connection is responsible for mainting the Server's (and hence Manager's) connection with a particular Slave. The Manager uses this Connection when sending a message to a Slave and also while receiving a message from a Slave.

Slave: The Slave class is also an executable and when run, establishes a connection with the Manager, and waits for commands from it. Upon receiving a command, the Slave determines the type of the command and responds appropriately. When a Slave receives a command to begin a process, the Slave creates a new thread for running that process. The Slave also maintains a timer to keep track of the status of all the processes that the Slave is running. Upon receiving a migration command, the Slave establishes a connection to the other Slave (to which or from the process has to be migrated).

1.1.2. Process path and process migration

The following is a step by step explanation of the process path:

1. The user types a start process command at the console of the Manager, specifying the process, the slave id of the Slave on which it wants to run the process and the arguments for the process.
2. The Manager instantiates an instance of the Process, assigns a unique process id to it, packs it into a message alongwith the arguments and other information needed by the Slave to run the process and uses the Connection class associated with the Slave socket to send the start process command to the Slave.
3. The Slave receives the message from the Manager (through the Connection) and creates a thread to run the assigned process with a given argument.
4. The Slave has a timer for checking the status of processes that it is running. Once it finds that a process has finished executing, it sends a process finish message to the Server (and thus Manager) which then notifies the user of the process running by displaying the related information on the console.

Assuming that a process with ID = x is running on Slave with ID = y. The following is a step by step explanation of process migration from the Slave with ID = y to Slave with ID = z :

1. The user types a migrate process command at the console of the Manager, specifying the process ID which it wants to migrate (in this case x), the Slave ID of the source Slave (in this case y) and the Slave ID of the target Slave (in this case z).
2. The Manager then sends migrate notify messages to both the Slaves (y and z), informing them of the migration. First, the Manager sends a message to the target Slave (z), so that the target Slave can start listening for a connection from the source Slave(y). Then the Manager sends a message to the source Slave(y) so that the source Slave can establish a connection to the target Slave(z).

3. Upon establishing a connection with the target Slave(z), the source Slave (y) suspends the process that has to be migrated. It then creates a thread to send this process (and other related information) to the target Slave (z).
4. The target Slave (z) receives the process (and other related information) from the source Slave (y) and creates a thread to resume the process. It then sends a migration complete message to the Manager which then notifies the user of successful migration.

1.2. Features

1.2.1. Health Monitoring

The Manager has a timer which periodically sends each Slave a health check message. It also maintains a health bar of each Slave (with max health = 5). Upon receiving a health check message, the Slave replies with a health check acknowledge message if it is alive. So, after the connection is established the Slave will have health = 5 (max health) meaning the connection is strong. However, if the Manager does not receive a health check acknowledge from a Slave in every check period, the health of that Slave is reduced by 1. If the health of a Slave reaches 0, it is assumed that the Slave has been disconnected from the Server and so the Slave is removed from the system. This is reported on the console to notify the user.

1.2.2. Concurrency

Every time a Slave receives a process start or process migrate command from the Manager (or Server), the Slave creates a thread to run the process or a thread to migrate the process. Thus, a Slave can concurrently migrate and run multiple processes.
Also, the Server maintains connections with all the Slaves in the system concurrently and thus, can interact with multiple Slaves concurrently.

1.2.3. Failure Report

Any time a failure occurs: while migration, starting processes or killing processes or a Slave goes down, the failure is communicated by the Slaves to the Manager which then notifies the user through the console.

1.3. Drawbacks

1.3.1. Slave Efficiency

In our design, the Slave still applies iterative handlers which means that one Slave can only deal with one command from the Manager at one time. So, for example, if it gets a start process command, it will create a thread to start the relevant process and only then will it be able to accept more commands from Manager. Thus, even though multiple process can be executing at the same time, only one process can be started at a time.

1.3.2. No Failure Management

The framework does not implement failure management. Thus, for example, if a process fails to start on some Slave, there is no mechanism for the Manager to attempt to start the process again on the same or different Slave. This failure to start the process is only reported to the Manager by the Slave. The Manager notifies the user of this failure but does nothing to fix it.

2. Build & Run

Follow these steps for building the project:

1. Import the project into an Eclipse workspace
2. Export the project as a Runnable Jar file from Eclipse

FOR CONVENIENCE, WE HAVE ALREADY PROVIDED THIS FILE WITH THE DELIVERABLES. IT IS CALLED "process_migration.jar".

Running the program requires 2 steps as well:

1. **Start the Manager program**: Type the following command on the command line of the machine where you want to run the manager program:
   java -cp <path_to_jar> main.Manager <server_port>
   <path_to_jar> is the path to the "process_migration.jar" file
   <server_port> is the port number for the manager to listen for incoming connections from slaves
2. **Start the Slave program:** Type the following command on the command line of the machine where you want to run the manager program:
   java -cp <path_to_jar> main.Slave <server_port> <server_IP> <slave_port>
   <path_to_jar> is the path to the "process_migration.jar" file
   <server_port> is the same port number that you used as <server_port> for starting the Manager program
   <server_IP> is the IP address (in "127.0.0.1" format) of the machine on which you started the Manager program
   <slave_port> is the port number assigned to the slave for connecting to other slaves

NOTE: You need to start the Manager program before you start any Slaves. Only one Manager program needs to be started, while many Slaves can be started on different machines. If running on AFS, make sure you start the program in the same directory on all machines so that file paths are consistent. Assign each slave a different port number.

3. Test

3.1. Commands that can be typed into the Manager console

ls : lists all the Slaves in the system
pn : lists all the process names that are recognized by the framework
ps : for each Slave in the system, lists all the processes running on it
start <process name> <slaveId> <args[]> : starts the process <process name> on the Slave with ID <slaveId> and <args[]> as arguments for the process
migrate <processId> <source slaveId> <target slaveId> : migrates the process with ID <processId> from Slave with ID <source slaveId> to Slave with ID <target slaveId>
kill <processId> <slaveId>: kills the process with ID <processId> running on Slave with ID <slaveId>
shutdown : shuts the Manager program down alongwith all the Slaves connected to it
help : lists all the commands that are recognized by the framework manager

3.2. Test cases

3.2.1. CountWordsProcess

What it does: Counts the total number of words in the given file, and for every line counts the number of words in that line
To run: **start CountWordsProcess <slaveID> <inFile> <outFile>**
<slaveID> is the ID of the Slave on which you want to run this process
<inFile> is the path to the file from which the process will read
<outFile> is the path to the file to which the process will write its output
NOTE: <inFile> must be an existing file. <outFile> need not be an existing file.

3.2.2. HeadProcess

What it does: Displays the first “x” lines of the given file, where “x” can be passed as an argument
To run: **start HeadProcess <slaveID> <num_lines> <inFile> <outFile>**
<slaveID> is the ID of the Slave on which you want to run this process
<num_lines> is the number of lines to display
<inFile> is the path to the file from which the process will read
<outFile> is the path to the file to which the process will write its output
NOTE: <inFile> must be an existing file. <outFile> need not be an existing file.

3.3. Example

1 Manager and 2 Slaves

First 2 slaves join with IDs 0 and 1, we start a ***CountWordsProcess*** with processID = 1 on slave0 as follows:

start CountWordsProcess 0 /afs/andrew.cmu.edu/usr2/lmao/t0.txt /afs/andrew.cmu.edu/usr2/lmao/out0.txt

and we start ***HeadProcess*** with processID = 2 on slave1 as follows:

start HeadProcess 1 10 /afs/andrew.cmu.edu/usr2/lmao/t1.txt /afs/andrew.cmu.edu/usr2/lmao/out1.txt

the slave0 will begin running the countwordsprocess and keep print the
“read lines %d total words %d”

before the slave0 finish this process we migrate to slave1 and we migrate HeadProcess to slave0

migrate 1 0 1
migrate 2 1 0

then fast to type ‘ps’ command we can see this process status will turn to migrating.
after migration is done we can see the continuous printing appear in slave1. Besides, if we type ‘ps’ again we can see this process is under slave1 and the status back to running again

then after running a few seconds, we migrate back this process to slave0
migrate 1 1 0

after the second migration is done and this process is still not finished. We try to kill this ***CountWordsProcess*** in the slave0 before it finished

kill 0 1

And at last we get the message from the slave0 that the ***HeadProcess*** is finished successfully, proving the commands we implemented is correct.