Migratable Process Framework
Final Report

Yiming Zong (Jimmy)
Carnegie Mellon University
Pittsburgh, PA 15213
yzong@cmu.edu

Abstract

This report reflects my final progress on Migratable Process Framework project for 15-440, Fall 2014. I will first survey and present the functionalities supported by the framework, then demonstrate the communication protocol between the Master and Child nodes. Also, I will mention the additional Migratable Processes supported by the framework and discuss the implementation of Transactional File I/O. Eventually, I will include building and testing instructions for my framework.

1 Functionalities Supported

1.1 Overview

The Migratable Process framework allows user to create, migrate, and kill jobs on denoted remote Child Nodes from the Master Node via a command-line interface. When a task is running, user is also able to migrate it from a Child Node to another without disrupting the state of the process. Last but not least, the user can also kill any running task on a given node or even terminate the Child Node completely. Following sub-sections contain more detailed description of the functionalities.

1.2 Master Command-Line Interface (CLI)

The Master Node comes with a command-line interface, where the user interacts with the Child Nodes and their processes. The prompt of the CLI looks like the following:

```
Process Manager @ 15440 > {user's command here...},
```

where `15440` stands for the port that Master Node listens to. Helpful success / error messages are returned to the user for each operation, and any Java exceptions (if occur) are displayed gracefully instead of as a full stack trace by default.

1.3 Child-Side Logging

Whenever Master Node interacts with Child Node, the Child logs the operation on `stdout`, preceded by their logging level: *Info*, *Warning*, and *Error*. Sample output after some node activities is as follows:

```
Starting up Child Manager at port 9899...
Info: Connected with Master Node jimmy-Thinkpad-T430:1341 as ID=node2.
Info: Created task GrepProcess with ID R4Bpv9D7.
Info: Handled REPORT request from Master.
Info: Handled REPORT request from Master.
Info: Resumed task GrepProcess with ID D1y6ozA4.
Info: Task with ID R4Bpv9D7 was PULLed by master node.
```

```
Warning: Master jimmy-Thinkpad-T430:1342 attempted to bind an occupied child node!
Error: Master Node requested RUN with unknown job name FooBarProcess!
Error: Master attempted to migrate nonexistent Task with ID 0xC0ffee!
Info: Successfully KILLALL and terminated Child Node.
#comment[<End of Output>]
```

1.4 Child Node Registration (`register`)

User on Master Node can link to new Child Nodes (started up separately) via CLI. The command usage is: `register <childHost> <childPort> <childId>`. Note that the `childId` field is a unique identifier for each Child Node, and it is stored on both the Master Node and the Child Node in concern. It will be used by the user to identify the Child Nodes in the future.

1.5 Poll Cluster Status (`list`)

User can use the command `list` to poll all Child Nodes that have been registered, along with their hostnames, port numbers, node ID, and active tasks. A sample output is as follows:

```
List of children and their jobs:

Child Hostname/Port: localhost:9898
Child Name: node1
Registered Master Hostname/Port: jimmy-Thinkpad-T430:1341
Active Job List: (Number of Jobs: 2)
         Task Id: WoVpfz1K       Task Name: GrepProcess
         Task Id: ms46c20A       Task Name: CopyProcess

Child Hostname/Port: localhost:9899
Child Name: node2
Registered Master Hostname/Port: jimmy-Thinkpad-T430:1341
Active Job List: (Number of Jobs: 1)
         Task Id: C3N4ErrM       Task Name: StatProcess
```

1.6 Run Migratable Process on Child (`run`)

To run a job on Child Node, user can use the `run <childId>` command and specify the task name along with required parameters. Each task, when created, is assigned a unique 8-digit alpha-numeric Task ID for future referral. A sample user interaction is as follows:

```
Process Manager @ 1341 > run node1
Run Job > CopyProcess /var/log/syslog.1 /home/jimmy/ds.tmp 1000
Job CopyProcess successfully created on Node node1.
```

Alternatively, if the task name (in the previous case `CopyProcess`) is not available on the child machine, an error message would be returned, saying "`Job creation failed! Reason: Job not found.`"

1.7 Migrate Process between Child Nodes (`transfer`)

User can transfer an active task from one Child Node to another by using the command `transfer <taskId> <srcChildId> <destChildId>`. If the given task ID is not found on the source node, an appropriate error message will be displayed to the user.

1.8 Terminate Child Process (`kill`)

User can terminate an active task on a Child Node by using the command `kill <childId> <taskId>`. This will update the active task list on the Child Node, and the action is safe and graceful (i.e. files are closed; resources are recycled).

1.9 Deregister Child Node (`dereg`)

User can choose to gracefully deregister a Child Node by using the command `dereg <childId>`. This will `kill` all active tasks on the node, close the `ServerSocket` on the Child Node, and eventually terminate the Child Node process to get rid of "orphans".

1.10 Graceful Master Termination (`exit/quit`)

User can quit the Process Manager CLI gracefully by using `exit` or `quit` command. This will attempt to terminate each registered Child Node by using `dereg`, then close the `ServerSocket` on the Master Node, and display status messages on the way like follows:

```
Process Manager @ 1341 > exit
Informing childs to exit...
Child Node 1 successfully disengaged and terminated.
Child Node 2 successfully disengaged and terminated.
Done! Goodbye.
<End of Output>
```

2 Communication Protocol

Overall, the Master Node and Child Nodes are organized in a master/slave model. Each Child Node has a persistent `ServerSocket` up and running on a separate thread to listen to package (`MasterToChildPackage` object) from the Master. The Child Node processes it by fulfilling the request, then send another package (`MasterToChildPackage` object) back to the master, providing requested information (like for `list`) or to acknowledge the completion of request (like for `kill`). To avoid concurrency issue, only one Master package is processed at a time. Following sub-subsections contain the back-end communication models in fulfilling the aforementioned commands from Master Node.

2.1 `MasterToChildPackage`

This package (implemented as a Java class) encodes a Master command and is sent to the Child Node for processing. It contains the following fields:

- `String command`: This field is similar to the `Method` field in HTTP requests. Its value can be `REG`, `REPORT`, `RUN`, `PUSH`, `PULL`, `KILL`, or `KILLALL`, corresponding to different use cases that will be explained below;
- `Object[] argv`: This is a generic parameter array. The length of array and the type of each element depends on the `command` field.

2.2 `ChildToMasterPackage`

This package (implemented as a Java class) encodes a response from Child Node being sent to Master Node. It contains the following fields:

- `String command`: This field is the same as the value of `command` in `MasterToChildPackage` for error-checking;
- `String status`: This field is the status code of the request. Its value can be `OK`, `ERR`, or `XXX`;
- `String message`: This field contains a textual description of the `status` field, and is especially useful when `status=ERR`. For example, it can be "Task not found" for a failed `KILL` request;
- `Object[] argv` (optional): This is a generic parameter array that holds extra data that the Child may sent to Master. Its value depends on the type of request.

2.3 Handling `register` Command

When a Child Node is run, it is an "orphan", without knowing any information about the Master Node. When a user runs `register <childHost> <childHost> <childId>` command on Master Node, it sends a package `(''REG'', [masterHost, masterPort, childId])` to the Child Node, such that the Child is now aware of the socket of Master Node.

If the operation is successful, the Child returns an acknowledgement package `(''REG'', ''OK'', ''OK'')` to the Master. Otherwise, it would return `(''REG'', ''ERR'', ''<errorMessage>'')`.

2.4 Handling `list` Command

When the Master Node polls for Child Nodes' status on user's demand, it sends a package `(''REPORT'', [])` to all registered Child Nodes, and each will return a package `(''REPORT'', ''OK'', ''<statusMessage>'')` to the Master. An example `statusMessage` would be: "*Child Name: nodeX\nActive Job List: (Number of Jobs: 1)\n\tTask Id: C3N4ErrM\tTask Name: GrepProcess*".

2.5 Handling `run` Command

When Master Node assigns a task to some Child Node, it sends to the Child a package `(''RUN'', [className, argv])`, where the `className` is the class name of the *Migratable Process*, and `argv` is the list of arguments for the process.

If the operations is successful, the Child returns an acknowledgement package `(''RUN'', ''OK'', ''OK'')` to the Master. Otherwise, it would return `(''REG'', ''ERR'', ''<errorMessage>'')`.

2.6 Handling `transfer` Command

When Master Node asks a task to be migrated from a Child Node to another, it is a two-step process in the back-end. First of all, the Master sends a package `(''PULL'', [taskId])` to the Child running the task. The child in turn suspends the task and sends its current state to the Master with a package `(''PULL'', ''OK'', ''OK'', [taskClassName, taskClassInstance])`. Then, the Master relays the task information by sending another package `(''PUSH'', [taskId, taskClassName, taskClassInstance])` to the destination Child Node and wait for acknowledgement.

If any step of the process above fails, an error message would be returned in the package (instead of "OK"), and the user on Master Node is notified.

2.7 Handling `kill` Command

When user at Master Node wants to terminate a specific task on a Child Node, the Master would send the package `(''KILL'', [taskId])` to the Child running the task. The Child then suspends the task, terminate the thread, and report the operation to the Master with `(''KILL'', ''OK'', ''OK'')`.

2.8 Handling `dereg` Command

When user wants to disengage and terminate a Child Node, Master would send a package `(''KILLALL'', [])` to the Child, which will suspend and kill all task threads on it then terminate the child daemon itself. An acknowledgement package `(''KILLALL'', ''OK'', ''OK'')` is sent back to the Master right before it terminates.

2.9 Handling `exit/quit` Command

This case is equivalent to running `dereg` on each registered Child Node, and then terminate the Master daemon itself.

3 Migratable Processes

In addition to the sample `GrepProcess`, I also included two more Migratable Processes, namely `CopyProcess` and `StatProcess`.

`CopyProcess` takes three arguments: `signature`, `inputFile`, and `outputFile`. What it does is that it copies each line in `inputFile` to `outputFile`, while preceding each line with its line number (stored as an instance variable), and appending each line with the `signature` string. The point of line number and *signature* string is to test that the instance variables of the job are preserved throughout migrations.

`StatProcess` takes two arguments: `lineNumber` and `outputFile`. What it does is that it will write `lineNumber` lines into `outputFile` every 300ms, each line consisting the host name on which the task is currently running on, along with a UTC timestamp. This offers a convenient way to tell at which Child Node a task is running at a certain time, especially when we `tail -f` the output file for live monitering. Also, this process intentionally lacks a `TransactionalFileInputStream` in order to verify that `MigratableProcess` can be flexible in terms of I/O.

4 Transactional File I/O Stream

Transactional File I/O Stream is utilized in all file I/O instances for this project in order to allow migrating processes with open files (including file location and file pointer). The Java classes for Transactional File I/O Stream are `TransactionalFileInputStream` and `TransactionalFileOutputStream`, extending `java.io.InputStream` and `java.io.OutputStream` respectively and supporting serialization.

Their underlying implementation is based on `java.io.RandomAccessFile`, which supports "file pointer" into any open file. This is desired because after process migration we need to set the file pointer to its previous location in order to pick up previous progress. For each operation (`read`, `write`), we essentially first seek to the file pointer stored as an instance variable in the Transactional I/O Stream Java class. Then, we carry out the key operation, i.e. `read` and `write`, update (increase) the file pointer based on the length of content read from / written to file, and eventually close the file. In this case, each `read` / `write` is a transaction, i.e. either data is read from / written to file *and* the file pointer is updated, or no file I/O is is done and file pointer remains intact.

5 Dependencies, Building, and Testing

Then `yzong` handin directory will contain two sub-directories, i.e. `reports` and `MigratableProcess`. In the former directory you can find this report file, and in the latter directory is the clean source code for the project.

5.1 Dependencies

This project uses two external libraries, Apache Commons CLI and Apache Commons Lang. The former one is used in handling the command-line arguments when running the Master and Child Nodes, and the latter is used to determine if a given string is a valid integer. Their jar files are included in the `lib/` directory of the Java project.

Also, to build the project from scratch without using IDE, an Ant Buildfile, `build.xml` is included in the root directory of the Java project. To use it, the machine needs to have Ant installed.

5.2 Building Instructions

To build the project from scratch, follow the steps below:

```
gkesden@ghc11 yzong$ cd MigratableProcess
gkesden@ghc11 yzong$ ls
    build.xml  lib  src
```

```
gkesden@ghc11 yzong/MigratableProcess$ ant build jar
    (Output omitted)
gkesden@ghc11 yzong/MigratableProcess$ ls
    bin  build.xml  lib  MigratableProcess.jar  src
```

To clean the project directory, run the following command:

```
gkesden@ghc11 yzong/MigratableProcess$ ant clean
    (Output omitted)
gkesden@ghc11 yzong/MigratableProcess$ ls
    build.xml  lib  src
```

5.3 Testing Instructions

The following test routine is designed to survey all functionalities of the Migratable Process framework. Feel free to experiment with them in different orders. The machines and ports in use in the sample are `ghc11:16441` (master), `ghc12:16442` (child), and `ghc13:16443` (child). Other combinations will also work, but be sure to substitute with the correct parameters below.

```
gkesden@ghc12 yzong/MigratableProcess$ java -jar MigratableProcess.jar -c -p 16442
        (Child Node started on ghc12. Monitor its stdout for useful logs.)

gkesden@ghc13 yzong/MigratableProcess$ java -jar MigratableProcess.jar -c -p 16443
        (Child Node started on ghc13. Monitor its stdout for useful logs.)

gkesden@ghc11 yzong/MigratableProcess$ java -jar MigratableProcess.jar -m -p 16441
Process Manager started at ghc11.ghc.andrew.cmu.edu:16441.
Process Manager @ 16441 > register ghc12.ghc.andrew.cmu.edu 16442 ghc12
Node registered.

Process Manager @ 16441 > register ghc13.ghc.andrew.cmu.edu 16443 ghc13
Node registered.

List of children and their jobs:

Child Hostname/Port: ghc12.ghc.andrew.cmu.edu:16442
Child Name: ghc12
Registered Master Hostname/Port: ghc11.ghc.andrew.cmu.edu:16441
Active Job List: (Number of Jobs: 0)

Child Hostname/Port: ghc13.ghc.andrew.cmu.edu:16443
Child Name: ghc13
Registered Master Hostname/Port: ghc11.ghc.andrew.cmu.edu:16441
Active Job List: (Number of Jobs: 0)

Process Manager @ 16441 > run ghc12
Run Job > GrepProcess status /afs/andrew.cmu.edu/usr18/yzong/public/dpkg.log ds01.out
Job GrepProcess successfully created on Node ghc12.

***While the job is running, 'tail -f ds01.out' to see live output.***

Process Manager @ 16441 > list
List of children and their jobs:

Child Hostname/Port: ghc12.ghc.andrew.cmu.edu:16442
Child Name: ghc12
Registered Master Hostname/Port: ghc11.ghc.andrew.cmu.edu:16441
Active Job List: (Number of Jobs: 1)
         Task Id: FsFMhfUx       Task Name: GrepProcess

Child Hostname/Port: ghc13.ghc.andrew.cmu.edu:16443
Child Name: ghc13
Registered Master Hostname/Port: ghc11.ghc.andrew.cmu.edu:16441
```

```
Active Job List: (Number of Jobs: 0)

Process Manager @ 16441 > transfer FsFMhfUx ghc12 ghc13
Suspending source job and migrating job...
Task info (FsFMhfUx) obtained from Node ghc12.
Job migration of FsFMhfUx successful from Node ghc12 to Node ghc13.

Process Manager @ 16441 > run ghc13
Run Job > StatProcess status ds02.out
Job GrepProcess successfully created on Node ghc13.

Process Manager @ 16441 > list
From the output, find the Task ID of the StatProcess we just created.

Process Manager @ 16441 > transfer hAGphbX5 ghc13 ghc12
Suspending source job and migrating job...
Task info (hAGphbX5) obtained from Node ghc13.
Job migration of hAGphbX5 successful from Node ghc13 to Node ghc12.

***While the job is running, 'tail -f ds02.out' to see live output.***
***Especially note the change of Host Name upon migration.***

Process Manager @ 16441 > run ghc13
Run Job > CopyProcess wahoo /afs/andrew.cmu.edu/usr18/yzong/public/dpkg.log ds03.out
Job GrepProcess successfully created on Node ghc13.

Process Manager @ 16441 > kill ghc13 cdaf7hK9
Killed task cdaf7hK9 from child ghc13.

***Output to file 'ds03.out' stops upon running 'kill' command.***

Process Manager @ 16441 > run ghc13
Run Job > CopyProcess wahoo /afs/andrew.cmu.edu/usr18/yzong/public/dpkg.log ds04.out
Job GrepProcess successfully created on Node ghc13.

Process Manager @ 16441 > dereg ghc13
Killed task cdaf7hK9 from child ghc13.

***'ghc13' node exits upon running 'dereg' command.***

Process Manager @ 16441 > exit
Informing childs to exit...
Child Node ghc12 successfully disengaged and terminated.
Done! Goodbye.

***Master Node terminates.***
```

6 Further Work & Enhancements

6.1 Status Reporting from Child Nodes

As described in Section 2.4, when a `list` command is issued from Master Node to Child Node, the child returns a *string* with status message of the node. Instead, a class `ChildNodeStatusPackage` can be defined to contain the Child Node's host name and port number, its Master's host name and port number, and also its job list. In this case, the Master can parse the data freely and use it in different ways.

Also, in order to eliminate orphans, Child Nodes can periodically send a heartbeat message to its Master, say, as (`''HEARTBEAT'', []`). If Master Node does not respond within five seconds, Child Node can believe that Master Node is down and thus can self-destruct.

6.2 Child Node's Return Value for `run`

As described in Section 2.5, after a Master Node sends a task to Child, it receives an acknowledgement message `(''RUN'', ''OK'', ''OK'')`. Instead, we can return the ID of the task, such that the Master does not need to perform a separate `list` operation to get it later on.

6.3 Making Communication Multi-Threaded/Asynchronous

Currently, all communications between Master and Child are sequential and synchronous, which means that the server thread of one party needs to wait for an acknowledgement from another in order to proceed sending/receiving another message. This can be changed by spawning a separate thread for handling each request; however, this can potentially lead to concurrency problem, so read/write locks and synchronizations might be needed. On the other hand, we can make the communication model entirely asynchronous (like in Amazon Web Services APIs), but that means we need to maintain a list of messages and their fulfillment status, which also leads to higher complexity.