15-440 (Fall 2014)

Lab 1: Portable, Migratable Work

Derek Tzeng

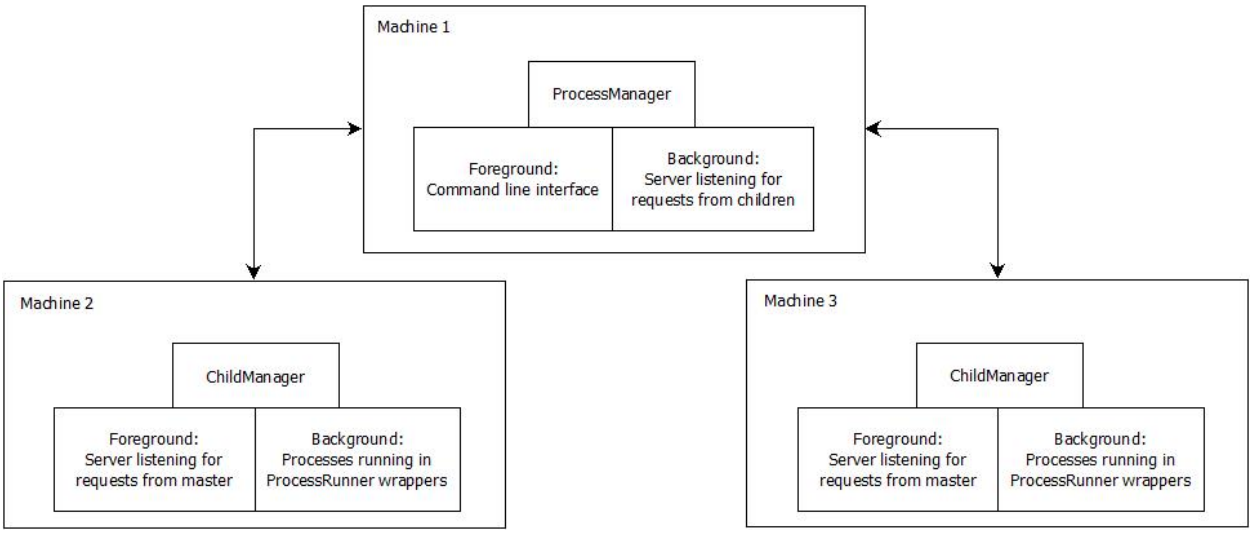
Features

Here is a documentation of all the features my framework provides.

- Master and child will automatically update state when necessary, user does not need to manually poll to refresh state
- For debugging, children provide simple logging to System.out when state of a process on the machine changes
- Command line interface that gives user ability to view state and manage processes

Command	Description
`run <child> <Class> <args[]>`	Runs a new process on the specified child according to the given class name and arguments
`suspend <name>`	Suspends the process with the given name. If the process was already suspended, this does nothing
`resume <name>`	Resumes the process with the given name. If the process was already running, this does nothing
`kill <name>`	Kills the process with the given name.
`migrate <name> <child>`	Migrates the process with the given name to the specified child. If the process was suspended before migration, it will resume execution once it reaches the destination child machine.
`list processes`	Lists the currently running processes (name, state, child)
`list children`	Lists the children nodes currently connected to the machine
`help`	Displays a help message that contains these commands
`quit`	Kills all processes and shuts down all child nodes, then exits

Design



My distributed framework is designed by following a master-slave architecture. A ProcessManager will run on the master node, while ChildManagers will run on the child nodes. Any migration of processes and communication to the ChildManagers will be done through the master (ProcessManager).

Now, I go into more detail about how my design works. First, a ProcessManager is started on the desired machine. This task will start up a background server and run the command line interface in the foreground. All user interaction will be done through this command line. The background server will listen for any new children or finished processes, so it can update its state appropriately.

After the ProcessManager is started up, the user can now start up any number of ChildManagers on the desired machines. The host and port of the master are supplied to the ChildManager on startup, so they will automatically establish a connection to the ProcessManager. If the connection to the ProcessManager is successful, the child will start up a server to listen for requests from the master.

Now that the distributed framework with the master and children has been established, the user can start allocating jobs to the children using the command line on the master. If the user types a command that requires interaction with a child, the master sends a packet to the appropriate child with the following fields (defined in the ProcessToChild class):

String command	String name	MigratableProcess process
“shutdown”	N/A	N/A
“run”	<process name>	<serialized process>
“suspend”	<process name>	N/A
“resume”	<process name>	N/A
“kill”	<process name>	N/A
“migrate”	<process name>	N/A

(Note: An “N/A” in a field means the field is not important for the corresponding command.)

The child will then receive this packet, do the necessary actions as specified, and send back a response packet with a boolean and MigratableProcess field (defined in the ProcessToChildResponse class). The boolean denotes whether the command requested has succeeded. The MigratableProcess field will only be used when a “migrate” command has successfully suspended the requested process and placed it in this field. The server will then parse this response and update its state if the command has succeeded.

On the other hand, the child may also need to initiate requests to the master. In our implementation, the request only has two different functions, so we have the following fields (defined in the ChildToProcess class):

String command	String name	String host	int port
“newchild”	N/A	<child hostname>	<child port>
“finished”	<process name>	N/A	N/A

The first command is used when a child is started up and wants to notify the master of its existence. The second command is used by our wrapper thread (ProcessRunner) that runs processes and waits for it to finish or be killed before notifying the master. For these requests, the server will just send back a boolean response of success or failure (ChildToProcessResponse).

To make migration of processes that use file I/O possible, I also implemented the TransactionalFileInputStream and TransactionalFileOutputStream classes. In these classes, I store the offset of bytes read/written, so any process that utilizes file I/O will not lose its offset when migrated to another machine.

Improvements

My design above has been fully implemented, but as always, improvements can be made. First of all, the exception handling in general can be improved. Most of my code handles exceptions in one large try-catch-finally block where I only catch the general “Exception e”. In addition to this, logging will also be beneficial. As a project like this scales in size, better exception handling and logging will be important factors for debugging possible issues.

Furthermore, my design does not deal with any hardware/network related issues. For example, if a TCP packet fails to send or a machine unexpectedly shuts down, my program could crash. Fixing this would complicate the code exponentially, as there would have to be even more exception handling and maybe even replication of data to protect against hardware failure.

Example Test Processes

I have implemented two example MigratableProcesses that conform to my interface. The first, CopyProcess, will copy the given file line by line into the output file. The second, LineCountProcess, will read the given file line by line and print out the corresponding line numbers into the output file. These two processes, as well as the example GrepProcess, will be used to run and test my framework below.

Building the Project

In my handin directory at "`/afs/andrew/course/15/440-f14/handin/lab1/dtzeng`", run the following script to build the jar:

```
chmod +x build.sh

./build.sh
```

This will create a jar file called "Migratable.jar". You may copy this jar, along with "short.txt" and "test.txt", to any desired directory for testing below.

Running the Examples

1. Find 4 GHC machines that are up and running. I will call these 4 machines GHC_A, GHC_B, GHC_C, GHC_D. Once logged in to these 4 machines, "cd" to the directory where the jar and text files were copied to above.

2. Truncate and tail the following files, so you can see the progress of the processes when they are started

```
echo -n "" > grepout.txt && tail -f grepout.txt

echo -n "" > copyout.txt && tail -f copyout.txt

echo -n "" > countout1.txt && tail -f countout1.txt

echo -n "" > countout2.txt && tail -f countout2.txt

echo -n "" > shortout.txt && tail -f shortout.txt
```

3. Start up the master process on GHC_A

```
java –cp <path to jar> migratable.managers.ProcessManager
```

4. Mark down the port number displayed in the command line prompt (`master@<port>`). I will call this PORT

5. Start the first child. On GHC_B,

```
java –cp <path to jar> migratable.managers.ChildManager GHC_A PORT
```

6. Start the second child. On GHC_C,

```
java –cp <path to jar> migratable.managers.ChildManager GHC_A PORT
```

7. Start the third child. On GHC_D,

```
java –cp <path to jar> migratable.managers.ChildManager GHC_A PORT
```

8. You can leave these 3 child machines running. All commands from now on will be done in the command line of the master process on GHC_A

9. Verify that all three children have been connected

```
list children
```

10. Run help to see a short summary of commands

```
help
```

11. Start a GrepProcess on GHC_B

```
run child0 GrepProcess grepper proc test.txt grepout.txt
```

12. Start a CopyProcess on GHC_C

```
run child1 CopyProcess copier test.txt copyout.txt
```

13. Start a LineCountProcess on GHC_C

```
run child1 LineCountProcess counter1 test.txt countout1.txt
```

14. Start a LineCountProcess on GHC_D

```
run child2 LineCountProcess counter2 test.txt countout2.txt
```

15. Try starting a process with the same name (should fail)

```
run child1 CopyProcess counter1 test.txt fail.txt
```

16. Try starting a process on a nonexistent child (should fail)

```
run child3 LineCountProcess failure test.txt fail.txt
```

17. Verify that the processes are running

```
list processes
```

18. Kill the GrepProcess

```
kill grepper
```

19. Suspend the CopyProcess

```
suspend copier
```

20. Verify that the state is updated correctly

```
list processes
```

21. Migrate the first LineCountProcess to GHC_D

```
migrate counter1 child2
```

22. Migrate the second LineCountProcess to GHC_B

```
migrate counter2 child0
```

23. Verify that the state is updated correctly

```
list processes
```

24. Start a short CopyProcess on GHC_C and immediately list processes

```
run child1 CopyProcess short short.txt shortout.txt
```

25. Wait for a little bit and run the list again. The CopyProcess should automatically remove itself when done, the other 3 should still be running (suspended for “copier”) as they are reading from a large file

```
list processes
```

26. Migrate the CopyProcess to GHC_D

```
migrate copier child2
```

27. Migrate the CopyProcess to GHC_B

```
migrate copier child0
```

28. Migrate the first LineCountProcess to GHC_C

```
migrate counter1 child1
```

29. Verify everything updated correctly

```
list processes
```

30. Kill the CopyProcess

```
kill copier
```

31. Verify.

```
list processes
```

32. Quit. Children should shut down.

```
quit
```