Lab 1: Process Migration

Kota, Yasaswy
ykota@andrew.cmu.edu

Venkataraman, Adwaith
adwaithv@andrew.cmu.edu

1 Implementation

The *ProcessManager* works by executing a process that implements *MigratableProcesses* on the node it is running on. A *MigratableProcess* is an interface that has a certain specification to it to ensure that all processes that implement it can be safely migrated by the *ProcessManager*. Each process that is started by a *ProcessManager* gets assigned a unique *ProcessID* (PID). The *ProcessManager* communicates with other instances of *ProcessManagers* that are on other nodes when a process needs to be migrated. All instances of *ProcessManger* will always *listen* to communication from other ProcessManager nodes to initiate migration.

Our implementation uses a Master - Slave topology to perform migration and synchronization. Before any slave can be started, a master node must be started. After that, any number of slaves can be launched. When a slave node first starts up, it communicates to the master node telling it that the newly started node is it's slave. The master node periodically polls all of it's slaves to make sure they are alive. An always updated list of the node list is maintained by the master, and is passed to a slave on-demand. This ensures that the slaves don't operate on stale data. This node list is saved in a *ConcurrentHashMap*, it stores revelant information such as each node's hostname, port and node ID.

When migration is to be performed, the user selects the process running on the current node to be migrated. The user also specifies where the process should be migrated. This will start the process of migration which involves putting the process in question into a known safe state called the *suspended* state. While the process is in it's suspended state, it is serialized and written into a file. The destination node will get a message after this is done and de-serialize the serialized object and start a new thread to run it. The program will continue from where it left off because the serialization will store the program state of the process. Interactions with files will also resume from where they left off because all file IO operations are done using the *TransactionalFileInputStream* and *TransactionalFileOuputStream* libraries which keep track of where the process is with regard to files it is interacting it with.

We use a *ConcurrentHashMap* to keep track of the running processes in a node. That data structure will be indexed by the unique ProcessID of a process. For it's value it will contain a custom object called *ProcessPacket* that contains that MigratableProcess object, the thread that runs the object, and the process's name and it's arguments.

We use polling instead of heartbeat to find out the status of running threads or slaves because a thread or slave can not always tell before it dies. Sometimes it if forcibly killed by a user. So polling from the server makes things more reliable.

No known bugs.

2 Compiling and Building

To compile and build, *cd* into the folder containing the **.java* files, the *Process-Manager* folder. In terminal, type the following commands:

```
1 $ make clean
2 $ make
```

This will produce **.class* files which is used for running the ProcessManager. A user *must* start the program first in *master* mode, then the user can create as many slaves as they desire. To start the program in master mode, type the following into the terminal:

```
1 $ java -cp . ProcessManager -m
```

The *PortNumber* the master will list on is hard coded to be 15000. The -m switch denotes it is a master.
To start a node in slave mode, assuming a server is started on unix1.andrew.cmu.edu:

```
$ java -cp . ProcessManager -p 15001 -hn unix1.andrew.cmu.edu
```

Here, the slave will list on port 15001. The -p switch specifies the port number and the -hn switch denotes the master's hostname.

3 Usage

The *ProcessManager* gives users the following commands to run on it.

- ps
- psall
- ns
- launch
- launchremote
- migrate
- migrateremote
- kill
- quit

3.1 ps

The *ps* command gives users the ability to see all the running processes in the node it is currently running on. As an illustration, you will see something like:

```
$ > ps
PID - Process Name (Arguments)
1   - test1(this)
2   - test2(is, an, example)
```

3.2 psall

The *psall* command gives users the ability to see all the running processes in the entire Master - Slave cluster setup.

```
$ > psall
Hostname: unix1.andrew.cmu.edu NodeID: 1
PID - Process Name (Arguments)
1   - test1(this)

Hostname: unix2.andrew.cmu.edu NodeID: 2
2   - test2(is, an, example)
```

3.3 ns

The *ns* command gives users the ability to see all the running nodes on the Master - Slave cluster setup. It also shows which of those is the master node and the node where the *ns* command is invoked.

```
$ > ns
NID - Port - HostName
1 - 15000 - unix1.andrew.cmu.edu ------master
2 - 15001 - unix2.andrew.cmu.edu ------current
3 - 15002 - unix3.andrew.cmu.edu
```

3.4 launch

The *launch* command gives users the ability to start new processes using the *ProcessManager*. In order to start a new process, the process in question needs to implement the *MigratableProcess* interface. The format to be followed is:
launch ProcessName Argument1 Argument2 ... ArgumentN

```
$ > launch test1 hello
$ > ps
PID - Process Name(Arguments)
1   - test1(hello)
```

3.5 launchremote

The *launchremote* command gives users the ability to start a new process on a remote node. In order to start a new process, the process in question needs to implement the *MigratableProcess* interface. The format to be followed is:
launchremote ProcessName Argument1 Argument2 ... ArgumentN DestinationNodeID

```
$ > launchremote test1 hello 3
$ > psall
Hostname: unix3.andrew.cmu.edu NodeID: 3
PID - Process Name(Arguments)
1   - test1(hello)
```

3.6 migrate

The *migrate* command gives users the ability to migrate a process from one node to another node that is running an instance of the *ProcessManager*. The format to be followed is:
migrate PID DestinationNodeID

```
$ > launch test1 hello
$ > ps
PID - Process Name(Arguments)
1   - test1(hello)
$ > migrate 1 2
$ > ps
$ > psall
Hostname: unix2.andrew.cmu.edu NodeID: 2
PID - Process Name(Arguments)
1   - test1(hello)
```

3.7 kill

The *kill* command gives users a way to stop a process that is running on the current node. The format to be followed is:
kill PID

```
$ > launch test1 hello
$ > ps
PID - Process Name(Arguments)
1   - test1(hello)
$ > kill 1
$ > ps
$ >
```

3.8 quit

The *quit* command gives users a graceful way to exit the *ProcessManager* that is running on the current node. If the *quit* command is initiated from the master, it kills all the slaves instances of *ProcessManager* first before exiting itself.

```
$ > quit
$
```

4 Examples

To show that the *ProcessManager* can handle IO operations seamlessly even with migration, we made 2 examples that interact with text files.

4.1 Example 1

For the first example, we will write a String that is stored within a program into a text file. The program goes through the length of the String while writing it one byte at a time. Since we want to show the migrational aspect of the *ProcessManager*, we have a *Sleep* function of 200ms within that write loop.

You can start the program by:

```
$ > launch test3
```

The program will start writing into the file named *filez.txt*. After a few seconds, the contents of the file will be:

```
$ cat filez.txt
Lorem Ipsum is simply dummy text of the printing and typesetting ind
```

Assuming another instance of the *ProcessManager* is running in unix2.andrew.cmu.edu and it is listening on port 15001 we can migrate the process:

```
$ > ps
PID - Process Name(Arguments)
1   - test3()
$ > migrate 2
$ > ps
$ > psall
Hostname: unix2.andrew.cmu.edu NodeID: 2
PID - Process Name(Arguments)
1   - test3()
```

We can check if filez.txt has updated.

```
$ cat filez.txt
Lorem Ipsum is simply dummy text of the printing and typesetting industry. Lorem Ipsum has bee
```

Thus we can see everything worked seamlessly.

4.2 Example 2

For the second example, we will read from a file (filez_read.txt) and write into another file (filez_write.txt). This read-write loop also has a *Sleep* of 200ms. The read file will contain an introductory paragraph on *Lorem Ipsum*.

The filez_read.txt will contain:
Lorem Ipsum is simply dummy text of the printing and typesetting industry. Lorem Ipsum has been the industry's standard dummy text ever since the 1500s, when an unknown printer took a galley of type and scrambled it to make a type specimen book. It has survived not only five centuries, but also the leap into electronic typesetting, remaining essentially unchanged. It was popularised in the 1960s with the release of Letraset sheets containing Lorem Ipsum passages, and more recently with desktop publishing software like Aldus PageMaker including versions of Lorem Ipsum.

```
$ > launch test4
$ > ps
PID - Process Name(Arguments)
1   - test4()
```

If we check the contents of the filez_write.txt file, it will contain:

```
$ cat filez_write.txt
Lorem Ipsum is simply d
```

Assuming another instance of the *ProcessManager* is running in unix2.andrew.cmu.edu and it is listening on port 15001 we can migrate the process:

```
$ > migrate 1 2
$ > ps
$ > psall
Hostname: unix2.andrew.cmu.edu NodeID: 2
PID - Process Name(Arguments)
1   - test4()
```

We can check if filez.txt has updated.

```
$ cat filez_write.txt
Lorem Ipsum is simply dummy text of the printing and typesetting industry. Lorem Ipsu
```

Thus we can see everything worked seamlessly.