Process Migration Report

Andrea Klein (andreakl), Yanan Jian (yjian)

1 Design

Using java, we have implemented a (master) ProcessManager class designed to communicate with any of a number of available Slave instances. Our framework supports four actions on any instance of a MigratableProcess: a process may be run, suspended, terminated, or migrated from one node to another.

2 Implementation Details

The core of our implementation consists of the following files:

1. ProcessManager.java - the key piece of code that manages the rest of the system. Receives command-line input from the user specifying the process, the action to be performed on that process, and the node(s) involved. Maintains the following:

   (a) a ConcurrentLinkedQueue<String> of IDs of the available slaves

   (b) a HashMap<String, String> of the IDs of the available slaves to their ip:ports

   (c) a HashMap<String, String> of suspended process ids to their dispatched slave ids

   (d) a HashMap<String, String> of newly launched process ids to their dispatched slave ids

   (e) a HashMap<String, String> of resumed process ids to their dispatched slave ids

   (f) once instance each of a MsgProcessor and a CmdProcessor, which run in separate threads

2. MsgProcessor.java - handles the communication of messages on the master's side. Maintains a socket, ObjectInputStream, and ObjectOutputStream for each slave.

3. CmdProcessor.java - prompts the user to input a command; enforces command formatting. Converts each valid command to a Msg and has it dispatched by the process manager.

4. Slave.java - when started up, a slave is given the IP of the master and the port it should use for communication with the master. It then sets up a socket accordingly, connects, and sends a greeting (essentially just to communicate its own existence) to the process manager. Thereafter, the slave listens to all the messages broadcast by the manager, and it processes those messages directed to it specifically. Processes are serialized and sent via messages between the master and slaves whenever necessary. The slave may be asked to perform one of the following actions on a process:

   (a) Launch - start the process in a new thread

   (b) Terminate - kill the process

   (c) Suspend - suspend the process, serialize it, and send it to the master in a Msg (in case the master wants to migrate it)

   (d) Resume - start the process locally in a new thread, after checking that it had previously been launched

   Note that a slave is essentially agnostic regarding the messages it receives. E.g. given an instruction to suspend a process, it doesn't know if the suspension is related to a migration or not; the response is the same.

5. Msg.java - provides a serializable class Msg used for communication between the various components of the framework. Includes attributes that identify the sender and intended recipient, any action desired by the sender, and any information the recipient may need to complete that action (for example, a MigratableProcess object).

6. MigratableProcess.java - general process interface that extends java.lang.Runnableand java.io.Serializable in order to ensure migratability. Any implementation is expected to include the void functions run(), suspend(), terminate(), and print().

7. TransactionalFileOutputStream.java / TransactionalFileInputStream.java - both extend InputStream / OutputStream and implement Serializable in order to permit the transmission of serializable objects between the components of the framework.

8. Constants.java - fixes the values of the master's IP and port numbers; defines the set of possible slave statuses {IDLE, SUSPENDED, TERMINATED, BUSY}, and defines the set of possible action attempts {SUCCESS, SID_NOT_MACHING_SLAVE, SLAVE_UNREACHABLE, IO_ERROR, UNKNOWN_ERROR, SID_NO_EXISTS, PROC_NOT_LAUNCHED}.

In addition, we will demonstrate our system on three classes that implement MigratableProcess: GrepProcess.java (provided), CountLines.java, and CpFile.java

3 Usage, Demos, and Testing

To use our framework:

1. From our submission directory, cd into the project directory: **cd src** (or cd /afs/andrew.cmu.edu/usr14/andreakl/DS_MP/s

2. Enter the command:

   ```
   /sbin/ifconfig $1 | grep "inet addr" | awk -F: '{print $2}' | awk '{print $1}'
   ```

   to get the ip address of the master (which will be first line of the output)

3. Compile the process manager: **javac ProcessManager.java**

4. Launch the process manager: **java ProcessManager**

For your convenience, instead of doing the last two steps yourself, you can also type **make pm**. Once the process manager is launched and awaiting input, you can go to the same folder on any afs machine and launch a slave. First compile the slave via **javac Slave.java**. Then, launch it with the command **java Slave -c 0.0.0.0:10000**, replacing 0.0.0.0 with the ip address of the master.

Get the ip address of the slave using the same command you used on the master. Switching back to the master node, you can now send commands, which the process manager will direct to a slave. You can try out different actions on the included sample text files, infile.txt and outfile.txt. (Note that infile.txt includes three lines, two of which include the string '640,' whereas outfile.txt starts out blank.) For example, **L GrepProcess 640 infile.txt outfile.txt** will grep for the string '640' in infile.txt and write any found instances to 'outfile.txt'. The following list includes examples of all the allowed commands.

1. **Launch a process:** L GrepProcess 640 infile.txt outfile.txt (will grep for the string '640' in infile.txt and write any found instances to 'outfile.txt')
   L CountLines infile.txt outfile.txt (counts the number of lines in infile.txt and writes the number to outfile.txt)
   L CpFile infile.txt outfile.txt (copies infile.txt to outfile.txt)
   Launching a process should generate the following kind of output on the master node (the one running the ProcessManager):

*****Running Processes:******
pid:[B@33bfc93a, ip:port:128.237.131.177:61740
****

2. **Suspend a process:** S [B@33bfc93a
   (This time we are using the pid as a suspend parameter, and [B@33bfc93a is the pid of a GrepProcess running on a slave. As promped from the command line, we use the pid as the identifier of each process, in case we want to launch multiple GrepProcesses on the same slave.)
   This should generate the following kind of output on the master node:
   *****Suspended Processes:******
   pid:[B@86e293a, ip:port:128.237.131.177:62928
   ****

3. **Resume a process on the same slave it is already running on:** R [B@33bfc93a

4. **Resume a process on some particular slave:** R [B@33bfc93a 128.237.131.177:61740
   This should generate the following kind of output on the master node:
   ****
   Resume:[B@33bfc93a
   ****
   (Note that the ip:port can only choose from the output ip:port above. This becomes important if you're running a slave and a master on the same machine. Although 127.0.0.1 and 0.0.0.0 are the same, as we record a slave ip as a String and generate a slaveid based on this String, if the master receives 127.0.0.1, then next time you want to resume a process on the slave, you have to input the ip address as 127.0.0.1 instead of 0.0.0.0.)

5. **Terminate a process:** T [B@33bfc93a
   This should generate the following output on the master node:
   *****Running Processes:******
   No Processes
   ****
   *****Suspended Processes:******
   No Processes
   ****

4 Robustness, Limitations, and Known Issues

The essential philosophy behind our design is to permit very flexible behavior by users of the process manager framework. That is, there are certain features that we do not explicitly implement, but that we have tried to make easy to implement using our framework. For example, our framework includes only lightweight internal load-balancing; however, it is designed for integration into a more sophisticated load-balancing program. In particular, it tracks the available slaves, their statuses, the processes on each slave node, and the statuses of those processes. While our process manager simply follows commands and does not try to redistribute work in any complex manner, it's easy to see how a load-balancing program might make use of that information. We note that the ability of the process manager to monitor the macrostate of the system might be improved by the introduction of some sort of polling system, which would check the status of the slaves at regular intervals (rather than just when messages are exchanged) tell it e.g. if a slave had died. In any case, we have intentionally delegated the monitoring of individual processes to the slaves themselves.

The process manager requires that commands be well-formed, and it prompts the user with a format reminder if it receives improper commandline input. However, more generally, our process manager does not try very hard to police the behavior of the user. This means that it does not always attempt to prevent the execution of impossible tasks, such as communicating with a slave that does not exist, or performing an action on a process that does not exist on the slave node to which the command was addressed. However, it reports back informative errors under such circumstances, rather than crashing.

Additional robust features include the following:

1. We have implemented an ID system for processes so that multiple processes of the same type may be run on the same node. For example, two GrepProcess instances may be run on the same slave node, as long as they are searching for different strings.
2. A slave may be run on the same node as the process manager.
3. Arbitrarily many slaves can be run on the same node.
4. If a slave suddenly dies when the master is sending a Msg to it, the master will get the error and will print a 'Slave does not exist' error message.
5. We prevent certain illegal operations such as suspending an unlaunched process, resuming an unsuspended process, etc.

Limitations of our framework include:

1. The process manager must be started before the slave node.
2. The state of the slaves is updated only upon exchange of messages (see discussion above). This means a ProcessManager may, upon receiving command-line input, try to make use of a slave that has died. Inputting the same command again may work, since the ProcessManager will then have an updated knowledge of the available slaves and can try to use a different one.
3. Multiple identical processes may not be run by the same slave.
4. A slave ID is generated based on the received ip and port, though 127.0.0.1 and 0.0.0.0 are the same, so the user has to input the exact ip that the master received. (We use ip:port as input argument to be user-friendly, i.e. so users do not have to remember the slave IDs)

We are not aware of any bugs in the included features.