Swapnil Pimpale (spimpale)
Romit Kudtarkar (rkudtark)

15-440/640 Project 1 Report

Collaboration Statement: All work submitted by us is our own. We did however, refer to the following github repository created by a former 15-440 student: https://github.com/ChiTheHotDogGuy11/Distributed-Systems/tree/master/440-HW1/src . However, we did not copy any code from the repository, we simply used it as a reference point for coming up with our initial design.

Introduction

This report contains information about the distributed systems framework we have written for project 1. Our framework runs a single *ProcessManager* on each node, with each node being able to talk to any other node within the cluster. We provide below instructions for operating and testing the framework, as well as an overview of the design of our framework.

Overview of Design

Our framework utilizes a many-to-many communication protocol, where any node in the cluster can send and receive commands from any other node in the cluster. Each node runs it's own *ProcessManager* and it's own *Server*. The *ProcessManager* is responsible for processing user input, sending requests to *Servers* on other nodes and receiving responses from other *Servers*. The *Server* processes incoming requests and sends the corresponding responses back to the requester. All requests and responses are encapsulated as objects, which are serialized and written over a socket. This allows us to re-use the same *RequestReceive* method in the *Server* class to read in *MigratableProcesses* objects or *Request* objects.

Features of the Framework:

- Detect thread completion: The framework is capable of detecting thread completion by installing listeners. The thread notifies it's listener of it's completion before dying.
- Support any process that adheres to the MigratableProcess interface (as stated below)
- Communicate over sockets: The framework supports communication between nodes using sockets
- Migrate/Kill processes from other nodes: The framework supports migrating a process from any node in the cluster to any other node in the cluster

- Add/remove hosts: The framework supports adding and removing unlimited hosts to and from the cluster
- List processes from all nodes: The framework can list running processes from all the nodes in the clusters

The MigratableProcess interface

The user wishing to run his/her own MigratableProcesses on this framework must extend the abstract class *MigratableProcess*. Additionally, any file I/O performed by the user-defined MigratableProcesses should only use the *TransactionalFileInputStream* and *TransactionalFileOutputStream* classes. In particular, the following conventions/protocols must be adhered to:

- If the user's MigratableProcess uses File I/O, the 'run' method should follow this convention:

```
try {

        while (!suspend_flag && !should_quit) {
                //do some work with files…
        }

inFile.closeFile();             //inFile is a TransactionalFileInputStream object
outFile.closeFile();   //outFile is a TransactionalFileOutputStream object
suspend_flag = false;

}  catch(Exception e) {
        …
}
finally {
        signalListeners();
}
```

- If the user's MigratableProcess does not use File I/O, the 'run' method should follow this convention:

```
        while (!suspend_flag && !should_quit) {
                //do some work…
        }
```

```
suspend_flag = false;
signalListeners();
```

For the file I/O case, the user's MigratableProcess must call the closeFile() method on any *TransactionalFileInputStream* or *TransactionalFileOutputStream* classes that it is using. This is because the *TransactionalFileInputStream* and *TransactionalFileOutputStream* classes do not close their streams at the end of their read/write methods. This enables those classes to avoid opening and closing streams each time a read or write is called. Once the *ProcessManager* calls suspend, the suspend_flag will be set true, and the user's MigratableProcess will break out of the loop and close the files before proceeding with the migration/termination.

TransactionalFileInputStream /TransactionalFileOutputStream classes

Note that the closeFile() method called on *TransactionalFileInputStream* or *TransactionalFileOutputStream* classes is a method defined by the framework, and not the traditional close() method defined by java's stream classes. If a user wishes to close the stream associated with either of these classes, the closeFile() method must be called.

Instructions for compiling and running our framework

1) Copy src directory over from handin.
2) Run the following commands:
    - javamkdir bin
    - c -d bin src/com/company/*.java
    - java -cp ./bin com.company.ProcessManager

The framework should now be running. The following commands are supported by our framework:

Command	Result of command
addhost <hostName>	Adds the specified host to the cluster. The host name must be specified in full, with no aliases.
removehost <hostName>	Remove the specified host from the cluster. The host name must be specified in full, with no aliases.

listhosts	Prints out all the hostnames in the cluster
<processname> [arg0] [arg1] ...	Start the specified process with the given arguments. Note that the process name is case-sensitive.
ps	Displays a list of running processes on all nodes of the cluster
migrate <processID> <fromHost> <toHost>	Migrate the process running on <fromHost> with process ID = <processID> to <toHost>.
kill <processID> <fromHost>	Kill the process running on <fromHost> with process ID = <processID>
help	Display information about commands supported by this framework.

To begin, start by running the framework on each node. Then, use any one node to add all the other nodes using the 'addhost' command.

For example, if I am running the framework on the following hosts:
angelshark.ics.cs.cmu.edu
bambooshark.ics.cs.cmu.edu
houndshark.ics.cs.cmu.edu
catshark.ics.cs.cmu.edu

Then I just run the following commands on the node at angelshark.ics.cs.cmu.edu:
addhost bambooshark.ics.cs.cmu.edu
addhost houndshark.ics.cs.cmu.edu
addhost catshark.ics.cs.cmu.edu

The result of this operation is that every node in the cluster is now connected to one another. The user may now proceed to start processes on each node, and migrate or kill them as desired.

Here is an example test of our framework:

- Step1: Compile and run the framework on all the desired nodes: angelshark.ics.cs.cmu.edu, catshark.ics.cs.cmu.edu and bambooshark.ics.cs.cmu.edu
- Step2: Add other nodes by using the addhost command on one node (i.e. run the following commands on angelshark.ics.cs.cmu.edu):
  addhost bambooshark.ics.cs.cmu.edu

addhost catshark.ics.cs.cmu.edu

- Step3: Use the ‘listhosts’ command to verify that all the hosts have been added correctly
- Step4: Start a process, say on angelshark.ics.cs.cmu.edu by running: CopyFileProcess <inputFile> <outputFile>
- Step5: Verify that the process has started using the ‘ps’ command
- Step6: Migrate the process from angelshark.ics.cs.cmu.edu to bambooshark.ics.cs.cmu.edu, by issuing the following command from the catshark.ics.cs.cmu.edu node:
  migrate 1 angelshark.ics.cs.cmu.edu bambooshark.ics.cs.cmu.edu
- Step7: Wait until the process finishes, and verify the correctness of the resulting output file.

What to do if a host goes down?

In the event that a host goes down, an appropriate error message should print upon trying to send a request to that host. The user can use the ‘listhosts’ command to get a listing of all the hosts in the cluster, and then use the ‘removehost’ command to remove the disconnected host. The user can also re-add the host later, if the host connection is revived. Note that the removehost command will still throw a socket exception when removing the ‘dead’ host, but the user can issue the ‘listhosts’ command to verify that the host is no longer in the cluster.

Testing the framework with CopyFileProcess.java and SortFileProcess.java

The two example processes we provided are CopyFileProcess and SortFileProcess. CopyFileProcess makes a copy of a file by reading in an input file byte by byte, and writing each byte to an output file. The SortFileProcess reads in an input file line by line, sorting the characters in each line, and writing the line to the output file.

The process can be started with their respective commands:
CopyFileProcess <inputfile> <outputfile>
SortFileProcess <inputfile> <outputfile>

We chose to implement these processes because they provided a simple but easy way to test the correctness of our migration command. For example, if we issued a command to migrate a CopyFileProcess from node 1 to node 2, and the resulting output file was not exactly the same as the original input file that the CopyFileProcess read from, we would know that there was a bug with our migration protocol. Migrating either one of these processes would require the framework to re-create the file handler and

restore the fileoffset before resuming the process on the new node. Both these processes work well for that purpose.