15-440/640 (Fall 2014)
Lab 1: Portable, Migratable Work
Due: Thursday, September 11th at 11:59pm

1. Introduction

One classic problem in distributed systems involves the migration of work. This problem is often referred to as that of *Process Migration*. But, the vehicle might be processes, objects, threads, or any other unit of active work. The common idea is moving in-progress work with as little disruption and wastage as possible.

To make this possible, the work-in-progress should be agnostic to what node it is running on and should be unaware of whether it has been moved from one node to another. Thus, it can enjoy the illusion that it is running on a single node during its entire execution.

To achieve this, it is necessary to be able to pause and package up a process, ship it to another node, and unpackage and resume it such that it is running again. The process should not lose it's location in the program, variables, open files, network connections or any other state.

In this lab, you will work with a partner to write a basic process migration system. In order to maintain a simple, consistent model throughout the handout, it is written with Java in mind. And, Java is a good choice, because it uses a ubiquitous virtual machine, uses an object-oriented model which provides a great way of decomposing problems into (possibly) migratable objects, and because it has strong, native support for serialization. Having said that, you are free – and encouraged – to work in the language of your choice.

2. Requirements/Deliverables

In this lab assignment you create a deliver for grading (i) a framework for migratable processes; (ii) a report which documents your design, implementation and testing, and tells us how to build and test your project; and (iii) test/example code, designed to show off the general-purpose and flexible nature of your design, that illustrates your framework working with at least two different migratable object types.

3.1 Migratable Processes

For this lab we will focus our attention on processes that are specially built to be migratable, which we'll refer to thdr as migratable processes. In order to simply the discussion, we are going to assume that the work is represented by object specially designed to be migratable, where the constraints are captured by an interface or abstract base class. We will refer to objects that implement such an interface or abstract base class as *MigratableProcess*es.

For those working Java, you'll likely want the *MigratableProcess* interface to extend *java.lang.Runnable*, such that it can be run via a *java.lang.Thread* object, and to additionally extend the *java.io.Serializable* interface, such that it can be serialized and written to or read from a stream (see

Section 3.6). Note that the recommended interface requires a *void* suspend(void) method which will be called before the object is serialized. This method affords an opportunity for the process to enter a known safe state.

A method that can produce a simple string representation of the object, such as by overriding Java's *String* toString(void) method, is also strongly recommended. This method can, for example, print the class name of the process as well as the original set of arguments with which it was called. Without this, debugging and tracing can be really painful.

It is recommended that *MigratableProcesses* include a constructor that takes, as it's sole argument, an array of strings. Doing this cleans up the interface, and is more likely to lead to a general-purpose framework than more complex options.

Finally, it is safe to assume that the process will limit it's I/O to files accessed via the *TransactionalFileInputStream* and *TransactionalFileOutputStream* classes, discussed in Section 3.2. In particular, other types of I/O might not be migration-safe.

3.2 Transactional I/O

To facilitate migrating processes with open files, you will need to implement a transactional I/O library. For those working in Java, you might want to implement *TransactionalFileInputStream* and *TransactionalFileOutputStream* classes which extend *java.io.InputStream and java.io.OutputStream*, respectively, as well as implement the *java.io.Serializable* interface.

When a read or write is requested via the library, it should open the file, seek to the requisite location, perform the operation, and close the file again. In this way, they will maintain all the information required in order to continue performing operations on the file, even if the process is transferred to another node. In order to improve performance, you can also choose to "cache" these connections by reusing them, unless a "migrated" flag is set, etc, in which case you wold set the flag upon migration and reset it any time a file handle is created or renewed.

Note that you may assume that all of the nodes share a common distributed filesystem, such as AFS, where all of the files to be accessed will be located. And, since you are writing the framework – you don't want to interrupt these methods with migration. Youj might find mutexes or monitors (Serialided objects and methods in Java) helpful for this.

3.3 Launching Processes

You should create a *ProcessManager* to monitor for requests to launch, remove, and migrate processes. You can have it poll to determine when processes die, receive periodic updates from them, and/or rely upon them to tell you as part of their depth. Think about the trade-offs of each method. Can a process always tell you before it does? What is the cost of polling or heartbeating?

When asking the *ProcessManager* to create new processes, you probably want it to accept (or return) a name for the instance. This way, you have a way to identify it later. Remember, it might be banaging several instance sof the same type at the saem time.

Your *ProcessManager* should be able to handle any *MigratableProcess* that conforms to your interface, abstract base class, and/or other requirements -- not just the examples you provided.

This means that you will not know what class you are instantiating until runtime. Thus, you will likely need to use something like Java's *java.lang.Class<T>* class and *java.lang.reflect.Constructor<T>* class to handle this at runtime.

4. Example

We've provided an example of a *MigratableProcess*, specifically a grep, for whatever value it might have to you.

5. Report

We should be able to grade your entire project by reading the report. Please follow the following format:

I. Clearly explain your design and illustrate its use, being sure to highlight any special features or abilities
II. Describe the portions of the design that are correctly implemented, that have bugs, and that remain unimplemented.
III. Tell us how to cleanly build, deploy, and run your project
IV. Highlight any dependencies and software or system requirements.
V. Tell us how to run and test your framework with your two examples.