**Lab 1: Portable, Migratable Work**

**Abhishek Bhowmik (abhowmi1), Neil Rajesh Dhruva (ndhruva)**

**Contents**

**Introduction …………………………………………………………………………….**

**Problem Definition …………………………………………………………………..**

**Solution Overview ……………………………………………………………………**

**Framework ………………………………………………………………………………**

**Code Organization ……………………………………………………………………**

**Conclusion ……………………………………………………………………………….**

**Introduction**

**Problem Definition**

The problem deals with building a distributed system that can migrate processes from one node to another, while preserving their state. The deliverables include:

1. Creating a framework for migratable processes.
2. This project report, which outlines the framework by providing information about the design and implementation of the system.
3. Test and example code for testing the system.

The distributed system must be able to preserve the state of file input/output operations, as well as the local state of the processes running on each of the nodes. It must also keep track of the various nodes that are part of the system.

Additionally, a user interface has to be provided which gives the user the freedom to:

1. Create a new process with a desired set of arguments on any of the active nodes.
2. Migrate a process form one node to another. The process has to restart from where it left off.
3. Get a list of processes and the information about the nodes they are running on.

**Solution Overview**

For creating a distributed system that preserves process state while migrating it from one node to another, we have created a model similar to the ‘Master/slave’ model. The master interacts with the user and gets the commands. It then informs a slave to perform those commands as per user request. Each request is processed one at a time.

The interaction between the master and user is limited to taking the command as an input, and printing reports of the current state of processes and slaves (nodes) to the user. Command execution is completely handled by the master and the slave(s), without involving the user.

The master-slave relationship is a bit more complex. Upon receiving the user’s command, the master determines if a slave needs to be instructed to either create a new process, or migrate an existing process. Based on this, the master creates a custom message that instructs the slave what action has to be performed. The slave decodes this message and performs the corresponding action. Additionally, the master periodically checks the status of each slave, i.e., whether it is still active (alive). The user can get this information through the appropriate command in the user interface. The master also provides user the information of which processes are still active on each of the slaves.

As mentioned before, the system also caters to preserving the state of a process while migrating it from one node to another. This includes the file input/output state of the process. Hence, when a process migrates from one node to another, it starts off exactly from where it left off.

With this general solution in mind, we have designed a framework that we describe in the next section to build such a distributed system.

**Framework**

**Code Organization**

The code has been organized in four directories within the src directory.

1. distsys.promigr.io - This is the I/O package that implements serializable I/O connections.
   1. TransactionalFileInputStream – generates an input stream through which the user can read a file. It also maintains the file offset so that when the user migrates the process from node A to node B, then node B starts reading the file from where A stopped reading it.
   2. TransactionalFileOutputStream – generates an output stream through which the user can write to a file. It also appends to a file if the user asks for that functionality.
2. distsys.promigr.manager – The manager package hosts classes that act as or aid the process manager (master) and the local manager (slave).
   1. ProcessManager – Acts as the master. It interacts with the user in order to take ‘create’, ‘migrate’ or ‘ps’ commands. For a create command, it creates the process and sends it to the node mentioned by the user for execution. In order to migrate a process, it instructs the corresponding node to migrate to the destination. Finally the ‘ps’ command lists all the processes in the system.
   2. LocalManager – Acts as the slave. It receives process creation and migration requests from the master and performs the corresponding action. Additionally, it generates a report of the processes that are currently running on it when the master asks for that information.
   3. ProcessManagerAssistant – Assists the process manager in keeping track of which processes are alive on the different nodes.
   4. LocalManagerThread – Handles the actual processing of user queries that are sent by the process manager to the local manager. This way, the local manager can cater to multiple queries without blocking.
   5. PollingRequestThread – Thread spawned by the process manager in order to keep track of the nodes that are alive.
   6. PollingResponseThread – Thread spawned by the local manager on each node to respond to the polling requests sent by the polling request thread, and ascertain that the local manager on the node is still alive.
   7. TableEntry – It serves as a bookkeeping table to keep track of the status of each process (whether active or not), the node that each process is running on and the process name and arguments associated with a particular process ID.
   8. ThreadObject – It keeps track of the MigratableProcess (explained in next package) object and thread associated with each process. This is useful when the local manager wants to migrate one of its processes to another node. It can help check if the thread running that process (object) is alive. Additionally, it can help suspend the thread for migration.
3. distsys.promigr.process – The process package consists of classes that aid process migration and message passing between two nodes on a network.
   1. MigratableProcess – It is an interface that every user class should implement in order to have the ability to migrate from one node to another without losing state.
   2. MessageWrap – It provides the ability to wrap the object corresponding to the user process, along with additional metadata for effective communication between the master and slave. The master can use it to send appropriate commands to be performed on the user process (object) that is wrapped within a MessageWrap object, which is then sent over the network.
4. distsys.promigr.test – This package consists of three test cases that can run on our distributed system.
   1. GrepProcess – It reads from a file, searches for a query and writes all the lines containing that query to another file.
   2. MergeFile – It reads three different files at once, and merges the files line-by-line into an output file.
   3. WebPageCopier – It read from a page on the world wide web, and copies the contents of the page to an output file.

**Conclusion**

The distributed system described above implements a multi-node master/slave framework that caters to a user’s commands of creating and migrating processes between nodes, as well as generating a report of the activity at any given time in the system. While the system works reasonably well according to the test cases that we performed, there are a lot of additions that can be performed (like ‘ack’) to make the system more robust.

However, we believe that this system can deal with requests from a single user efficiently, and also perform efficient and frequent bookkeeping in order to generate useful reports for the user.