15-440/640 Distributed Systems

Lab 1 Report: Portable, Migratable Work

Team:
Abhishek Bhowmik (abhowmi1),
Neil Rajesh Dhruva (ndhruva)

Date:
09/11/2014

Contents

Problem Definition ……. 3

Solution Overview …… 4

Framework …… 6

Code Organization …… 11

Conclusion …… 13

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.

As opposed to a peer-to-peer model, the master/slave model provides better control over the system. While it could create a situation of a single point of failure, it brings in more control for one global manager (master) to run different tasks on different machines, and also keep track of each of these tasks. Additionally, the global manager can keep track of all the local managers (slaves) and their status. Finally, it provides an efficient interface to interact with the user in order to process user queries.

In 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.

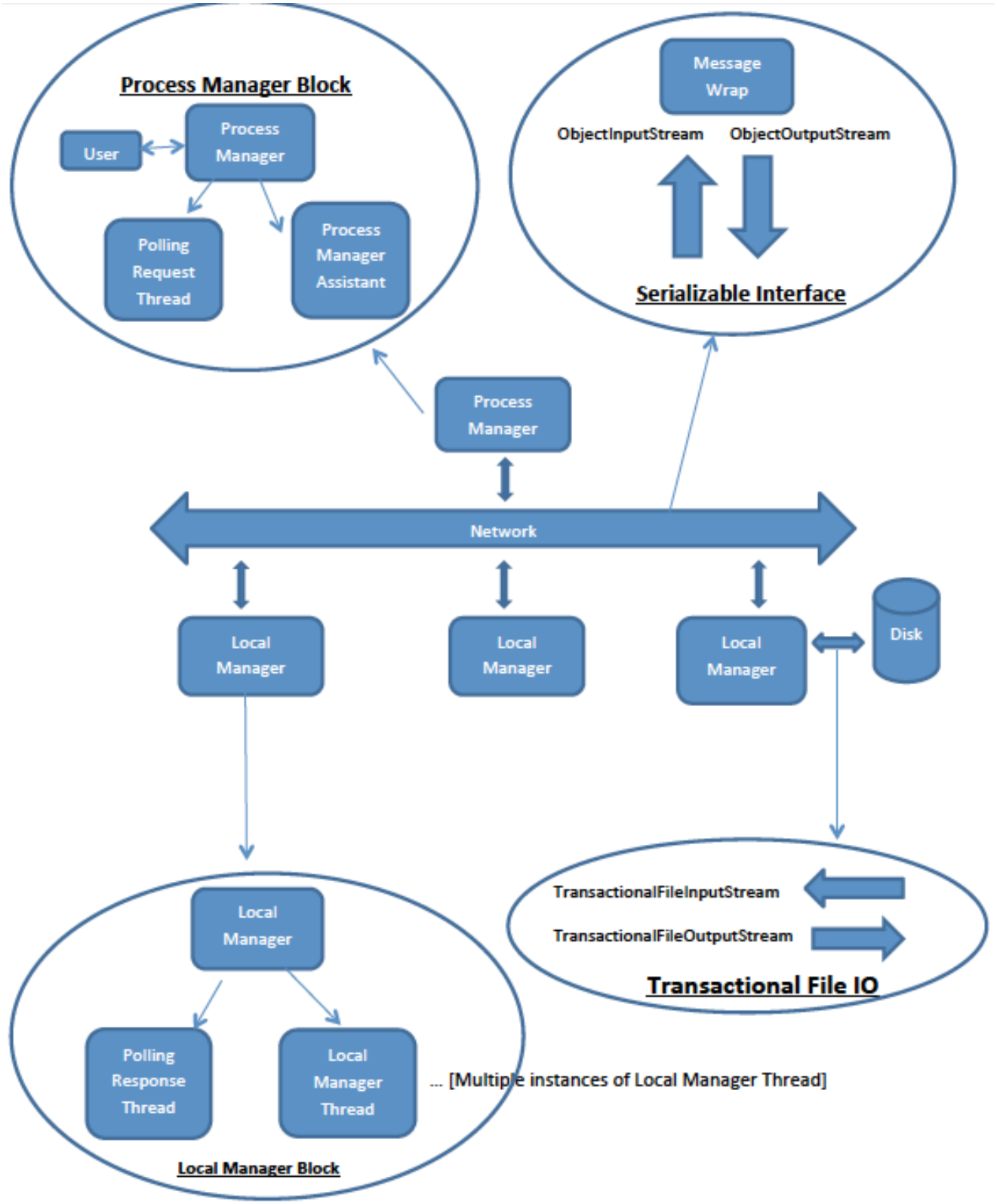
The master-slave relationship is a bit more complex. Upon receiving the user's command, the master determines if a slave needs to be instructed to either create a new process, or migrate an existing process. Based on this, the master creates a custom message that instructs the slave what action has to 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


Process Manager Block
User
Process Manager
Polling Request Thread
Process Manager Assistant
Message Wrap
ObjectInputStream
ObjectOutputStream
Serializable Interface
Process Manager
Network
Local Manager
Local Manager
Local Manager
Disk
TransactionalFileInputStream
TransactionalFileOutputStream
Transactional File IO
Local Manager
Polling Response Thread
Local Manager Thread
... [Multiple instances of Local Manager Thread]
Local Manager Block

As indicated in the block diagram, we have four major components in our system (enclosed by ovals), namely:

1. Process Manager Block
2. Serializable Interface
3. Transactional IO
4. Local Manager Block

We now explain the function and interfaces of each of these blocks separately.

1. **Process Manager Block**: The Process Manager Block is the master for the whole system and hence there is only one instance of this block. This block has three sub-components, namely – ProcessManager, ProcessManagerAssistant and PollingRequestThread.

   The ProcessManager prints a prompt to the screen and accepts the commands from the user. The different types of commands include:

   a. create: creates a new process on a particular node (defined by the user). It also generates a unique process ID to identify the process. The user is expected to use this for future references to the process.
   b. migrate: migrates a process from one node to another, and starts the process on the new node.
   c. ps: lists the currently active processes along with node names and arguments.
   d. help: provides the command menu.
   e. exit: exits the process manager.

   It also maintains a list of active machines, and the list of all active processes in the distributed system. It launches the user-requested process on the appropriate LocalManager. The ProcessManager also spawns a ProcessManagerAssistant thread, which serves the purpose of listening to responses from LocalManager machines. This is helpful in the event of a 'ps' command request from the user. On getting such a request, we query all the active machines on the system and wait for the responses in the ProcessManagerAssistant thread.

The reason we run a separate thread for listening is because we want to set up the listening server port on the master before we send the queries to slave machines. If we do this in the ProcessManager, it will block, waiting for slave connections and not allow us to go ahead with querying the nodes.

Additionally, in order to check if the nodes (slaves) are still alive, the PollingRequestThread polls the slave machines every 2.5 seconds. We use polling instead of heartbeats to monitor node activity because it gives us the flexibility to query all the nodes at once and observe the established connections. Additionally, heartbeating would queue the messages from different nodes and process them one by one (else use more system resources to process them simultaneously). However, heartbeating would conserve some time and network resources on the slave nodes. Nonetheless, for a small-scale system, polling or heartbeating will not result ideal implementation would be to use heartbeating coupled with polling.

2. **Local Manager Block**: The Local Manager Block represents an instance of a slave. It is responsible for listening to requests for launching new process, for every such request it spawns a new LocalManagerThread, which then takes the action of creating a new process or migrating to new node as requested. It sends its responses back to ProcessManagerAssistant in the Process Manager Block. The Local Manager Block also has a PollingResponseThread that listens to polling requests from the ProcessManagerAssistant.

   The communication between the local manager and the process manager is established through sockets that listen on specific ports.

3. **TransactionalFileIO:** This block is responsible for carrying out file IO. It has two components, namely TransactionalFileInputStream and TransactionalFileOutputStream, which carry out read and write operations on a file, and maintain the state associated with each

operation. This is helpful in the case when we want to interrupt a file operation, in order for it to be migrated and resume from the point where the operation was interrupted.

The state is saved using a global state variable in each of the IO classes. Additionally, each of the two classes is serializable and hence can be serialized as part of the user process (object).

Our framework makes use of the underlying distributed Andrew File System (AFS) in order to read and write to files available on the system. Hence, during migration, it is not necessary to migrate the file. Instead, we can migrate only a reference to a file and start read/writing from where we left off.

4. **Serializable Interface:** This block represents the object streams over which we can read and write serialized byte stream representations of MigratableProcesses. This is done with the help of messages of type MessageWrap, which contain metadata about the action to be performed with the associated with the MigratableProcess.

   Object serialization is the key to migrate a process from one node (local manager) to another. We use this functionality provided by Java in a variety of situations ranging form transfer of messages to the transfer of user processes (objects).

Dividing the system design into these 4 groups made it easier to integrate the system at the end. Each group functions independently, and interacts efficiently with the other groups through reliable communication. Hence, we now have a functioning integrated distributed system to cater to user requests.

Possible Additions

A few possible additions to the framework could include:

1. Acknowledgement messages from local managers about successfully starting or migrating processes. This has been addressed to a great extent by:
    a. Creating the process object (without running in thread) in the ProcessManager and then sending it to the desired node.
    b. Checking if the destination (node B) of the migrate command is alive before sending the migrate signal to the corresponding local manager (node A).

2. Implementing a combination of heartbeating and polling to use the least resources (system and network) on the master as well as the slaves.

3. Even more thorough error checking, and error reporting from slaves to master.

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.
    a. `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.
    b. `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).
    a. `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.
    b. `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.
    c. `ProcessManagerAssistant` – Assists the process manager in keeping track of which processes are alive on the different nodes.
    d. `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.

e. `PollingRequestThread` – Thread spawned by the process manager in order to keep track of the nodes that are alive.
f. `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.
g. `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.
h. `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. Also, 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.
    a. `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.
    b. `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.
    a. `GrepProcess` – It reads from a file, searches for a query and writes all the lines containing that query to another file.
    b. `MergeFile` – It reads three different files at once, and merges the files line-by-line into an output file.
    c. `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.