Report of ProcessMigration
Project 1 of 15-640
Kailiang Chen(kailianc) & Yang Pan(yangpan)

Design

In this project, our goal is to design and implement a tiny framework, which realizes migration of work in distributed systems with as little disruption and wastage as possible. We refer the feature as **Process Migration**, however, the work might be processes, objects, threads, or any other unit of active work. As shown in Figure 1, **Process Migration** is a code plus data software migration used in Mobility field. Its procedure is to pause and package up a process, ship it to another node, and unpackage and resume it such that it is running again.

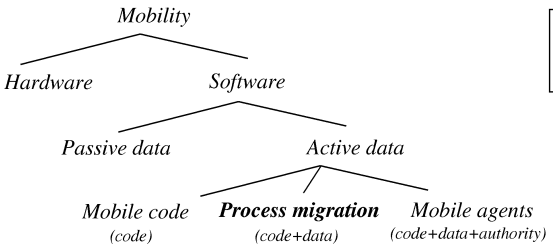


Figure 1

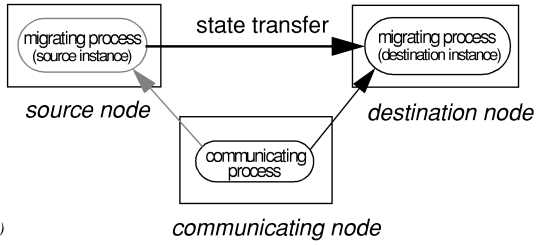


Figure 2

Our system is designed and implemented using Java, which is a very suitable language for distributed system programming as it provides persistence (Serialization/De-serialization) to package/unpackage objects, Socket library API for network communication to send/receive packaged object stream, Concurrency mechanism to synchronize shared resources access between threads, and Reflection to remotely invoke method of unpackaged objects.

The basic user case of our system is: (See Figure 2)

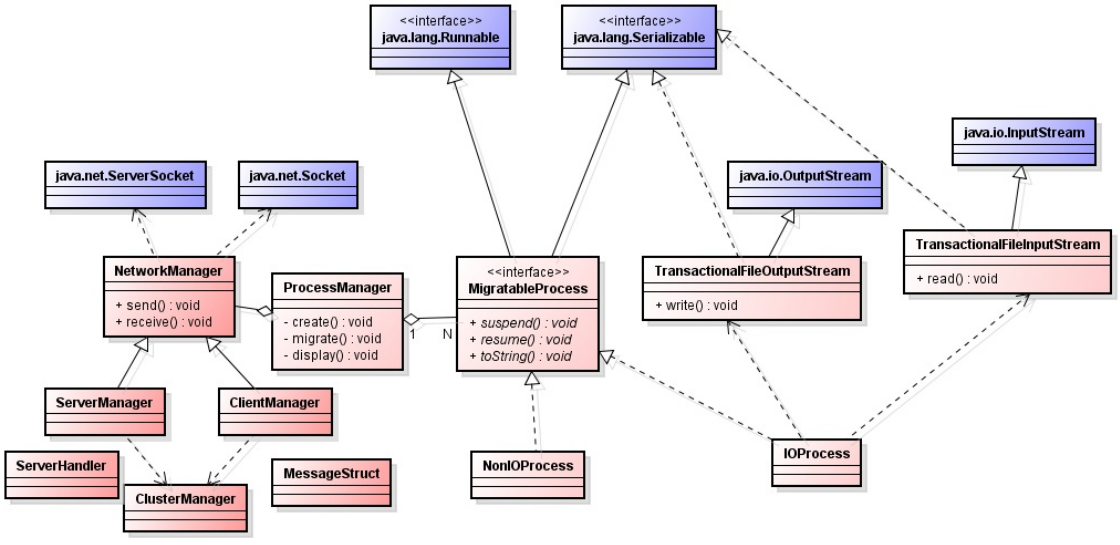
1) A communicating process is launched on a node (Master) to coordinate load information management and distribute scheduling between Slave machines.
2) A source process is launched on a source node (Slave A) to do some heavy work.
3) By some load balance policy, the Master finds another significantly reduced load destination node (Slave B) and give a request to Slave A to migrate process to Slave B.
4) Then the source process is suspended on Slave A, and serialized object stream is sent to Slave B from Slave A.

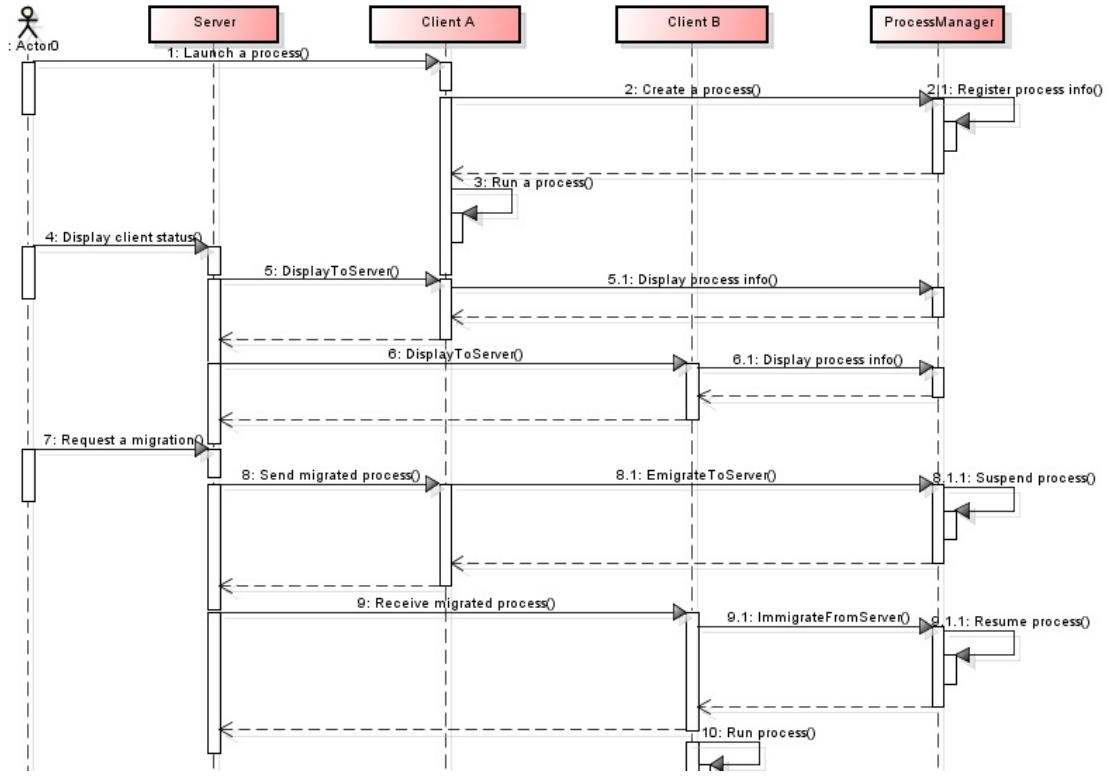
5) The Slave B receives the stream, deserializes and resumes the source process.

Considering the complexity of Process Context (Stack, memory and file management), Thread is used to simulate Process in our system. And the detailed load balance management computation, which might be based on the utilization of CPU or the length of process queue waiting to be executed is omitted here, a simple command message is used instead. However, on each Client, process information such as process id, process name can be displayed by “ps” command, meanwhile, all the client process information can be displayed on Server. Based on the information, it can be determined which process should be migrated from a source node to which destination node.

Furthermore, our system uses the Receiver-Initiated policy for distributed scheduling instead of Sender-Initiated one, because the Receiver-Initiated policy is easy to control and implement for high-load systems with many overloaded nodes and few underloaded ones, and is especially suitable for Process Migration.

UML





Special features

1. Multiple clients supported. One server can check the status of all clients and can control the process migration between them.
2. Process information check on client. You can check out the process information on every client.
3. Function call to migratable process on demand. You can invoke a function of a running migratable process with an input command.
4. Network communication scalable. There are generalized message structure and dispatcher in the framework. You can add any type of message to the framework conveniently.
5. Load balancer friendly. The control manager class (ClusterManager) is loosely decoupled from all other parts in server side. You can replace it with your own load balancer.

Implementation

Class Implementation

According to the following UML graph, our system consists of:

1) MigratableProcess: an interface to provide migratable features for classes implementing it. It defines the interface that suspend/resume function has to be overrided during migration.
2) IOProcess/NonIOProcess : Concrete classes implemented MigratableProcess interface to give samples for Process Migration. One of our examples is a Process involves IO operation by use of TransactionIO libray to read shuffled alphabets, sort them and write to a new file, the other is a Process works as a simple counter involving no IO operation. Of course, it is possible that multiple instances of the same type running on the node and one of them is to be migrated. Actually the user case has been tested.
3) ProcessManager: a comprehensive class to monitor for request to launch, remove and migrate processes. When the creation of a new process instance is requested, a new process id is also created to identify the process. And it can also handle any classes that implemented MigratableProcess by instantiating until runtime based on Reflection.
4) TransactionFileInputStream/TransactionFileOutputStream: a transactional IO libarary for maintaining all the information to continue operations on the file even if process is migrated to another node(machine). A migrated flag is created to select reusing the file connection or not, which is realized by setting the flag upon Migration and resetting it any time a file handler is created or renewed. Synchronization is used to avoid interrupting file read/write operation during migration.
5) NetworkManager/ClusterManager/ServerManager/ClientManager/MessageStruct/ServerHandler: a server /client library to simulate distributed system environment by use of Sockets, multi-clients load balance is controlled and adjusted by a server, which receives process migration request, finds a good candidate and transmits byte streams of serialized objects from one client to another.

In Transaction IO and Process part, various Concurrency methods are used to ensure thread-safety. Read/write operations are synchronized to avoid resource access collision incurred from multiple instances of the same process type, and the coordination between io operation and migration is also considered. In ProcessManager class, ConcurrentSkipListMap is used to associate the

ProcessID with Process instance as well as provide the thread-safety, while AtomicInteger type gives the Atomicity and volatile keyword ensures the Visibility of a variable in Concurrency operations.

In network communication part, ClusterManager and ServerManager are designed to simulate a Server node to control migration between Client nodes. ClientManager can be instantiated multiple times to simulate multiple Client nodes. Communication between Server and Multiple Clients is realized by Socket based on a Message Dispatch mechanism. Requesting of Process Migration, Command to overloaded/underloaded Clients, Send/Receive of serialized object stream are all controlled by Server.

Development environment

This project is developed with Eclipse IDE for Java Developers, Luna Release (4.4.0), JDK 8u20. If you want to write a test class inheriting MigratableProcess, you should work in the same environment.

Test with our examples

We provide two process classes to test the TransactionIO classes and other migration features. They are NonIOProcess and IOProcess.

NonIOProcess is a simple class with only a counter tick-tocking every 0.1s. After migration, the counter starts from the stage right before migration.

IOProcess mainly test the TransactionIO classes, TransactionalFileInputStream and TransactionalFileOutputStream. This class keeps reading in a series of shuffled characters from a file, sorting them and writing them out to a file.

Test environment

A server at unix.andrew.cmu.edu, unix machines connected to AFS

Deploy and run

1. Open three Terminals and connect them to a server connected to AFS

using ssh. Make sure they have the same login user (saving the troubles brought by file permission).

2. Change directory to ./src/. (for convenience, we denote ProcessMigration/ as ./) in three Terminals.
3. Type “make” and then “make run” in the three Terminals.
4. In one Terminal, type “server” to become a server. And you will get an echo showing the IP and port (say, 192.168.0.1:6777). In the other two, type “client 192.168.0.1 6777) and they will connect to that server.
5. In Client A, type “create NonIOProcess” to create a migratable process without IO operations.
6. In Client B, type “create IOProcess input.txt output.txt” to create a migratable process with IO operations. Note: input.txt and output.txt should have absolute path.
7. In server, type “ps” to show all the running processes on all clients. In the two clients, type “ps” to show all the local running processes.
8. In server, type “migrate 1 0 0” to migrate the IO process on Client B (cid 1) to Client A (cid 0).
9. In server, type “migrate 0 0 1” to migrate the non-IO process on Client A to Client B.
10. In server, type “ps” to show all the running process on all clients after migration. In the two clients, type “ps” to show all the local running processes.
11. Wait for IOProcess to finish on Client A. When it shows “JOB COMPLETED”, the IOProcess finishes all the IO operations.
12. In server, type “exit” to exit server. All other clients connected to this server will exit automatically.

Understand the test result

The result consist of three parts:

1. Migration status: In step 7, you can see the NonIOProcess running on Client A and IOProcess on Client B. In step 10, you can see NonIOProcess running on Client B and IOProcess on Client A. This means that the two processes have been migrated to each other client.
2. Process status: In step 9, we can see an echo from NonIOProcess saying its counter, like “NonIOProcess : suspend(), cnt = xxx” on Client A. After migration, we can see an echo like “NonIOProcess : run() begin, cnt = yyy”, where yyy = xxx + 1. This means the suspended work before migration is resumed after migration.
3. IO status: In IOProcess, we read a shuffled alphabet one character by one

character from input.txt, sort them and write them to output.txt, and repeat this procedure 8 times. So, open output.txt, and you can see 8 set of ordered alphabets in it. This means the IO operations works fine during migration.

Writing customized test class

By implementing the MigratableProcess class, you can write your own process class to test TransactionInputFileStream, TransactionOutputFileStream and other migration features.

Developer environment

You can write the class anywhere you want, as long as you inherit from MigratableProcess and implement the three abstract methods. However, it's strongly recommended to develop in the same environment as ours.

Understand our interfaces and framework

To run your test class under our framework, you must inherit your test class from MigratableProcess. In MigratableProcess, there are three abstract classes that you have to override in your own class:

1. suspend(): This method will be called before the object is serialized. It affords an opportunity for the process to enter a known safe state.
2. resume():This method will be called after migration. Resume all the work that was suspended.
3. toString():This method is used for debugging. It can print the class name of the process as well as the original set of arguments with which it was called.

Besides, here is some other things should be noted: (assume your class file is GregProcess.java)

1. The constructor of your test class should have exactly one argument with type String[], no matter you need it or not.
2. You can design some test methods to be called after GregProcess has been instantiated. This method should have exactly one argument with type String[].
3. At the beginning of GregProcess.java, you should declare the package by typing "package edu.cmu.andrew.ds.ps;".
4. In the suspend() method

Deploy and run

Once you finish your work, copy your test class file (GregProcess.java) to the directory “./src/edu/cmu/andrew/ds/ps/”. And then, edit ./src/Makefile by adding “./edu/cmu/andrew/ds/ps/GregProcess.java” to “CLASSES”.

Now, change directory to ./src/ and type the following commands to run the whole program:

```
> make
> make run
```

In the client, after you connect to a server, you can type the following command to create a new instance of your process:

```
create GregProcess [ANY OPTIONAL ARGUMENT TO CONSTRUCTOR]
```

You can also call any method in GregProcess to test by typing:

```
call PID METHOD_NAME [ANY OPTIONAL ARGUMENT]
```

where PID is the pid of that instance.

Use all these interfaces to test our framework.