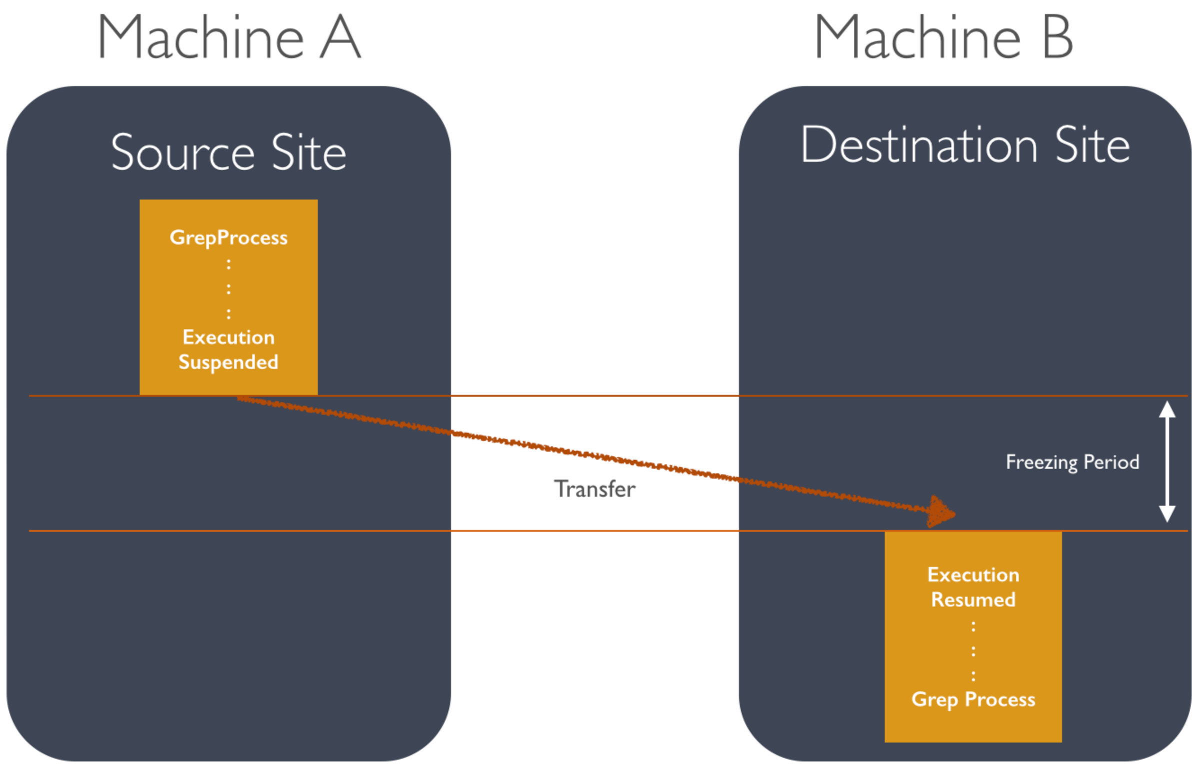


Concept borrowed from: Prof. Munehiro Fukuda (University of Washington)

Process Migration Framework

Prepared by: Abhishek Sharma, Summer 2014

SEAMLESS PROCESS MIGRATION FRAMEWORK

ARCHITECTURE, DESIGN, IMPLEMENTATION AND DEPLOYMENT

1. Architectural Design + Usage + Special Features

Architecture and Design

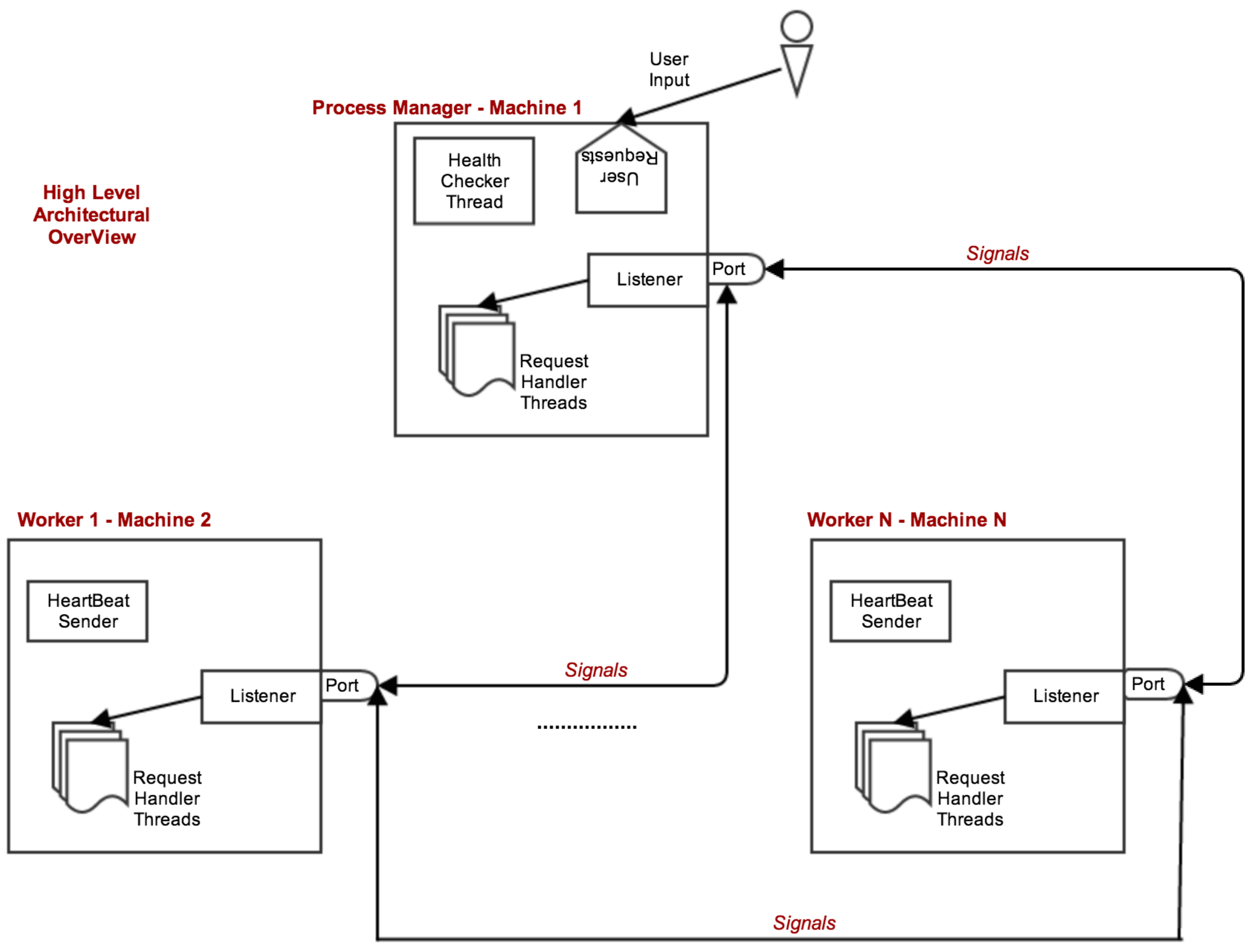
The architecture of the system was primarily motivated around on four key principles:

a. Seamless Work Migration (Application User Experience) & Work State Preservation: The central idea was to migrate processes (unit of work) cleanly without affecting the actual work that is being done. Irrespective of the process migration from machine x to machine n; as an end-user running the some application process; the process migration should be seamless. The process must begin precisely at the same location when it stopped executing (state must be preserved) and another key aspect was interrupting the process cleanly; since all the core logic was in a while loop we would wait for the process to finish one cycle of doing whatever it was doing and then cleanly suspend the user process. As an administrator one should be able to monitor and move process around very easily with commands.

b. Ability for the software to be used as framework as long as the advocated policy was followed: Since this software had to built as a framework it meant that any process that was Migratable Process would be allowed to migrated over this framework. There should be runtime detection of what kind of processes are asked to be launched, migrated or stopped. *(As pointed out by one of the TAs)* if the this framework would be used at an industrial scale level (with thousands of process to migrate) the process migration framework must be easily adaptable possibly to an automated load balancing mechanism that would run alongside to orchestrate the migration process based on the load on each of the machines.

c. Multi-Threaded Workers and Process Managers is order to support high-availability in event of simultaneous requests and failures: There would multi-directional communication happening across all the different machines in the distributed system which requires listeners on each machine that listen s to request and spawn handler threads to address those multiple requests. This is true for the process manager and all the worker machines in the systems.

d. A commonly understood policy/mechanism for message exchange between everyone in the distributed system: The central theme here was to define a singular mechanism/format for message exchange. This format would need to be understood by all machines in the distributed system and should have the ability to be easily packed and unpacked. Different information would need to be packed for different kinds of signals such all machine would be able take the appropriate action on receipt of the signal.



Key Design Decisions

Structural Design: The Process Manager caters to user input and also maintains state about the health of all workers (aliveness and list of running processes) in-order to display it. Workers are entities (machines) that are essentially running the user processes and listening/sending to requests to one another. They also received requests from process manager to start processes. It is important to note that the Process Manager machine could also possibly a worker in the distributed system. Process Manager acts a central mechanism for delegating user commands/control but apart from that the whole system should work in a peer-to-peer sort of mechanism.

Communication Mechanism: As per a key design principle discussed above about having a common agreed policy of communication amongst the processes I designed a Signal Factory. Based on the user input a particular type (launch, migrate etc) would be manufactured and would be sent across the wire to another machine. The Signal object would also encapsulate all the information that needs to be send across the network example object state of the running process or launch command. This provided for a singular mechanism for encoding and decoding information. Thus, the signal object would be serialized and transferred over the wire with the necessary information; then unserialized at destination to do the needful.

Choice of Heartbeat as opposed to Ping-Echo Mechanism: I heartbeat mechanism to report the status from various machine in the system. Choosing the heartbeat versus ping-echo was a tricky decision in system design. They key factors we considered while making this decision were:

a. Deadline at which device status information should be reported (Criticality of Operation).

b. The load on the communication channel.

Heartbeat requires 1 message over channel while Ping and Echo would required 2 messages. Since we have to implement a monitoring interface at the end and under the assumption that network guarantees deliver of messages I decided to promote overall system responsiveness by doing heartbeat. Also, with ping-echo the Process Manager would need to have a special mechanism to broadcast and ask which process are reporting to me to sort of get feedback from newly instantiated workers. On the contrary the heartbeat mechanism provides mechanism of registering with Process Manager as soon as they come into existence. The heartbeat mechanism promises better confidence on service level agreements. Although this did mean that I would have more messages on the communication channel but I accepted since so that I could promoted more robustness in the system. This begs the question what if a user issues a command on a worker that is dead; I managed this basically with the strategic setting of timing of HealthChecker running on process manager and HeartBeat sender thread on Workers.

Transactional I/O: Considering that we could assume a distributed file-system the Transactional I/O implementation was based on saved offset of the current point of operation (reading or writing a file). This meant that even if a process was suspended this information would be passed along to the next machine it would shift the offset to the correct position and resume normal operation.

Listeners and Request Handlers: Each machine runs a server socket that always keeps listening for any communication request. Once a request is received it spawns of thread of a respective request handler that caters of the request. This way the listeners are never blocked and the overall system is responsive to messages floating around in the system.

Concurrency Issues Management: The Process Manager and Workers all maintain internal Map of relevant/necessary information. In this distributed system workers and/or process's die or new come alive again all the time. This would imply that multiple requests for things could potentially hit simultaneously and this would cause concurrent access to shared data structures leading to an exception or bad state. I handled this by ensuring that only one thread at a given time has access to the shared Map.

Special Features or Abilities

1. Robustness of system: after the process is suspended and is about to be migrated if the network fails and the migrating request cannot be sent then the work restarts on the same machine again. The idea is never to stop the work.
2. The Signal factory that could easily be extended to add additional kinds of signal or information easily while keeping the policy still the same.

2. Portions of Design that are correctly implemented, have bugs, and that are unimplemented

Portions of Design Implemented	Have Bugs	UnImplemented
All of the above discussed design decisions and considerations have been implemented	Based on my testing I tried to fixed all the bugs I could possibly find. At the instance I did not find any bugs that I am aware off. Although: I have not handled all the scenarios from aspect of robustness meaning there a million ways you which a person might wrong command to mess with a system. I have not handled all those cases.	Not Applicable

3. Process to build, deploy and run the project

1. Download the source code and *cd* to *SeamlessProcessMigration_DistributedSystem*

2. Compile the system by running the following command:
```
    make
```

3. To start a Process Manager run the following command:
```
    make processmanager IP=10.0.0.2 PORT=5555
```

4. To start a Worker run the command:
```
    make worker IP=10.0.0.2 PORT=9000 PMIP=10.0.0.2 PMPORT=5555
```

5. To start the GrepProcess follow the instructions on the Process Manager prompt. Example below:
```
abhi.ds.GrepProcess <query> <inputfile> <outputfile> -location 10.0.0.2:6665
```

6. To start Factorial process follow the instructions on the Process Manager prompt. Example below:
```
abhi.ds.Factorial <Integer> <outputfile> -location 10.0.0.2:6665
```

7. To print a List of all Workers; use the following command:
```
listallWorkers
```

8. To print a List of all running processes on a particular Worker; use the following command:

```
listallps <IPAddress:Port>
```

9. To Migrate a process from one machine to another; use the following command:

```
<ProcessID> -fromLocation <IPAddress:Port> -toLocation <IPAddress:Port>
```

4. Software Dependencies

The process migration framework has been build using the Java 1.7. There are no external dependencies or additional software requirements. The namespace of the package is **abhi.ds.**

5. Testing the Framework with Two Examples

Two MigratableProcesses are implemented for testing.

1) *GrepProcess*. This class implements a similar function as unix command 'grep'. If one line of the input file contains the pattern, it will be written to output file.

This process takes three arguments: *pattern, input file path, output file path.*

Example: => abhi.ds.GrepProcess abhi <path>/1.txt <path>/2.txt

2) *Factorial*. The class calculates the Factorial of an integer and then writes it to an output file. It's a basic factorial calculator.

This process take two arguments: *integer, output file path*

Example: => abhi.ds.Factorial 8 <path>/xyz.txt