Darrin Willis - dswillis
Bohan Li - bohanl

15-440 Writeup
Lab 1: Migratable Processes

https://github.com/darrinwillis/distributedSystems

Overview

The overall design of our project consists of a centralized Delegation Server, which allows clients to request processes to be added, and then subsequently dispatches processes to clients, which then run the processes in a Process Manager. The entire project is written in Java, with a little bit of bash scripting to ease compilation and running.

Migratable Process

A migratable process is specified by extending the `MigratableProcess` interface. This only includes the ability to `run()` or `suspend()` itself. Migratable Processes are stored and shifted around as files, which are written and read by using the `ObjectInputStream` and `ObjectOutputStream` classes respectively, which are abstracted away and handled by static functions in a `ProcessIO` class. The file name is uniquely determined by a “process ID” number which is determined by the Delegation Server, and is then passed to the client to determine which process it should run. This implementation relies on the client already having access to the specified file.

Delegation Server

The delegation server has an exposed RMI interface under the name of `MasterServerInterface`. This interface allows any client to connect and `register` itself, which then causes the server to begin allocating a portion of its list of processes to the new client. The server does not directly communicate with the ProcessManager, instead operating through a `ProcessManagerClient` with an RMI enabled `ProcessManagerClientInterface`. This client class handles all of the networking and acts as a barrier from the `ProcessManager`.

The server takes in a `Class <? extends MigratableProcess>` as an argument in order to add processes, which it instantiates using the `Constructor` class, failing if the arguments which are passed to it do not match any constructor for that class.

The server accepts new clients and maintains a list of the currently viable clients. Every second the server does three things: assign processes, load balance, and update process list.

AssignProcesses checks for differences between the process list the server has, and the processes that are currently running. Then it syncs the differences by adding processes to the first client for load balancing.

For load balancing, we first obtain an average load by counting the processes running on each client, making sure to catch any concurrency and connection exceptions. Our load balancing argument takes any client that is some BALANCE_LEVEL higher than average and migrates some of its processes to other clients. The victim clients are selected at random out of all current clients. Once we finish balancing, we send a request to each client with their new processes.

UpdateProcessList checks if the file associated with the process has been deleted by a suspension. If it has, it updates the server's list of processes.

Also, our server contacts all clients to retrieve their list of currently running process. If the client cannot be reached, then the server removes the client from the list of viable clients. This could potentially result in duplicate work if the client was disallowed from contacting the server, but still enabled to reach AFS. The closest we were able to test this was by starting a server and client and then killing the server, at which point the client continued to run all processes to completion.

Process Manager

After the Delegation Server contacts the client to tell it to set its list of processes, the client forwards it onto the Process Manager. This manages the threads, maintaining a link between each process and thread. It can take a list of process names from the client and run/suspend processes until it matches the received list. The first time this occurs, a thread checker is starting in the background that checks and deletes finished threads. The process manager can return its list of running processes

It also supports the suspending and unsuspending of processes. These read/write the process to input and output streams. When unsuspending a process, it also starts a thread for it.

Transactional IO

Makes connections to input and output streams in order to open files, seek to an index, perform and operation, then close the file. It does not cache these connections or set migrated flags.

Compiling / Running

Simply run `make` to compile all necessary code. `make clean` to clean out all `.class` files.

```
./startServer.sh
```

starts the server on a given machine.

```
./startClient.sh rmi://server.url.com
```

connects a client to the server at the given url (Default url is unix12.andrew.cmu.edu)

```
./startTestingClient.sh rmi://server.url.com
```

connects a testing client to add 10 instances of `GrepProcess` to the server (Same default url)

Testing

Much of testing on this framework has been done in a heavily white-boxed manner, rather than black-boxed testing. We have specifically tweaked the inputs to test various edge cases.

When given any number of clients and no processes, the server simply holds onto its clients and waits until it receives a process to send off.

When given any number of processes without any clients, the server maintains its processes until a viable client connects.

When given a significantly greater number of processes than clients, the server distributes the processes among them.

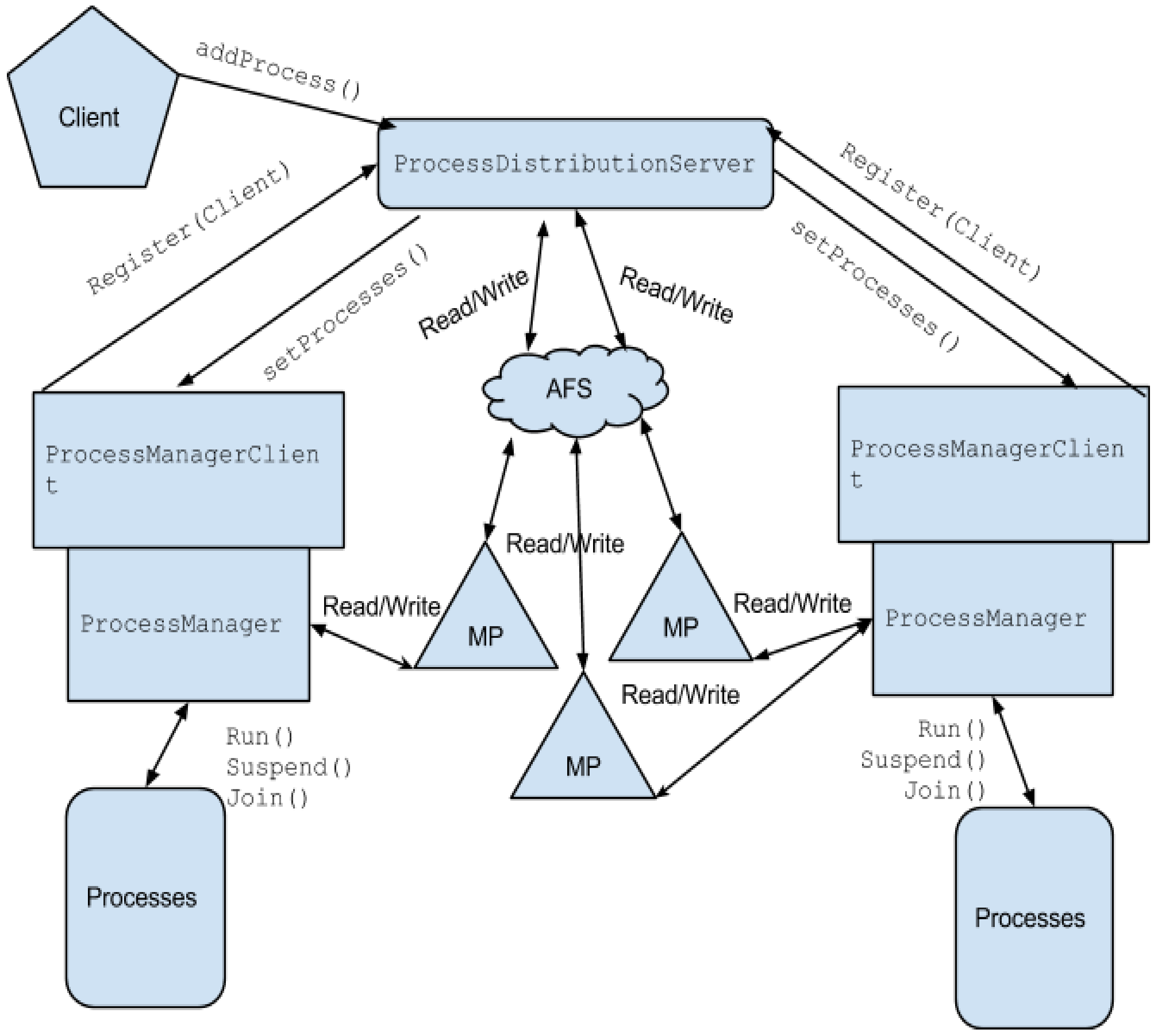
Potential Issues and Possible Improvements

There seems to be a rare, non-reproducible bug or race condition in which the server stops correctly setting processes. This has only occurred once in a later build, so it is unknown if it has been totally resolved.

Processes are only written to file when `suspend()` is called, so it is possible for clients to repeatedly die and not write the process state to file, leading to duplicated work (but correct output).

TransactionalIO could use some sort of caching.

Darrin Willis - dswillis
Bohan Li - bohanl


Client
addProcess()
ProcessDistributionServer
Register(Client)
setProcesses()
Register(Client)
setProcesses()
Read/Write
Read/Write
AFS
ProcessManagerClient
ProcessManagerClient
Read/Write
Read/Write
Read/Write
Read/Write
Read/Write
MP
MP
MP
ProcessManager
ProcessManager
Run()
Suspend()
Join()
Run()
Suspend()
Join()
Processes
Processes