SWM49 Distributed Systems Principles

Post-coursework assignment, Part II

Mike Pountney, Student ID: 09843038

Table of Contents

Assignment Outline

Your task is to investigate the possibility of creating a layer of software, which would run on the top of existing Operating Systems in a distributed nature. You are expected to design one such a system and to implement some functionality of it. Your design should be oriented towards multiple processes running and communicating in that environment. There is no special requirement what processes should do, but it is expected that they run in indefinite loop. The system should be able to provide for some the following:

- [R1] To recognise how many processes are running.
- [R2] To establish that some of the processes crashed and to re-establish the group.
- [R3] To choose a coordinator if one dies.
- [R4] To implement a distributed mutual exclusion mechanism.
- [R5] To accept a new process in the group if one is started from a command line by a user.
- [R6] To provide a web interface so that a user may get some statistics about the system.
- [R7] To provide for anything else that you may wish to add which is relevant to the system.

Design Summary

Given the assignment outline, I have chosen to implement the multi-process distributed system with an asynchronous architecture using Java Messaging Service (JMS) and ActiveMQ.

Since there is no requirement for what the processes should do, I have simulated a task with `Thread.delay(5000)`, which at will block the task thread for five seconds - as a normal process would do, for example if shelled out to restart an operating system service, or to perform heavy disk IO.

The overall design principle with task allocation is that **Tasks** are submitted to the messaging system 'tasks topic' using some form of client (for example, using the **TaskSubmitter** Java app), with an optional 'lock required' parameter. If a **Lock** is **not** required, then the task is executed concurrently by all the '**ClientProcess**' task runners. If a lock **is** required, then the task is executed sequentially by the 'ClientProcess' task runners, with the locking handled by a ClientProcess that has granted itself '**Controller**' (lock-manager) state. Client processes can only execute if a **lock request** sent to the messaging system is acknowledged by the controller with a **lock granted** message. Upon completion of the task, the client with the lock releases it with a **lock unlocked** message, which the controller uses to

In the event that a lock request goes unanswered for a period of time it is assumed that the controller has died, and an **Election** is called by that process. It announces via the messaging system that it **requests to promote itself** to be the controller. Every other process in the system will interpret this request and, if it's own ID (randomly chosen on startup) is higher, it will start its own election request. If a process with an active election request receives an election message from a process with a higher ID, it accepts that it has lost the election and silently drops its request. If a process does not receive any election messages from processes with higher IDs, then after a time-out period it assumes that it has the highest ID, and **announces** that it is the new controller (promoting itself in the meantime). If a node with a higher ID receives this announcement (due to being offline during the election, or other error), another election is called.

Supporting Code

The code supporting this design is available to browse and download via the following URL:

https://github.com/mikepea/swm49-project

Design Choices

The choice to use ActiveMQ and JMS was driven by [R5], [R3], and [R4]:

- As the clients do not depend on each other, accepting a new client process into the group is straightforward, it simply becomes another listener to the JMS topics - satisfying [R5].
- As the clients do not need to connect directly with one another, communicating with the controller and mutex mechanisms is straightforward in the event of the re-election of the controller - new connection details do not need to be passed around - simplifying [R3] and [R4].
- ActiveMQ has fault-tolerant and network-of-brokers modes, which can be used to expand the network of processes worldwide. [R7]
- ActiveMQ also provides a comprehensive Admin interface, which allows us to see the number of processes (consumers) - satisfying [R1] - and message statistics (somewhat satisfying [R6])
- [R2] is satisfied in the controller case by the re-election process. It is unnecessary in the general process case due to the loose-coupling of the processes themselves.

Rather than use JMS ObjectMessages and passing Java Objects (POJOs) through the messaging system, I have chosen to serialise the objects using JSON ('Javascript Object Notation') and pass them as simple JMS TextMessages. This was done to ensure that the objects could be created and read from any language, not tying the platform to Java alone [R7]. This is discussed further in 'Future Enhancements'.

Detailed Design

There are two applications provided:

- The '**ClientProcess**' class - this attaches to the message queues, and listens for incoming tasks
- The '**TaskSubmitter**' class - this process can be used to submit tasks onto the task queue.

The TaskSubmitter is essentially reference code for submitting tasks, it simply forms a suitable Task object, serializes it, and puts it onto the ApacheMQ 'tasks' topic.

ClientProcess is the core 'task processing' process. This has the following facets:

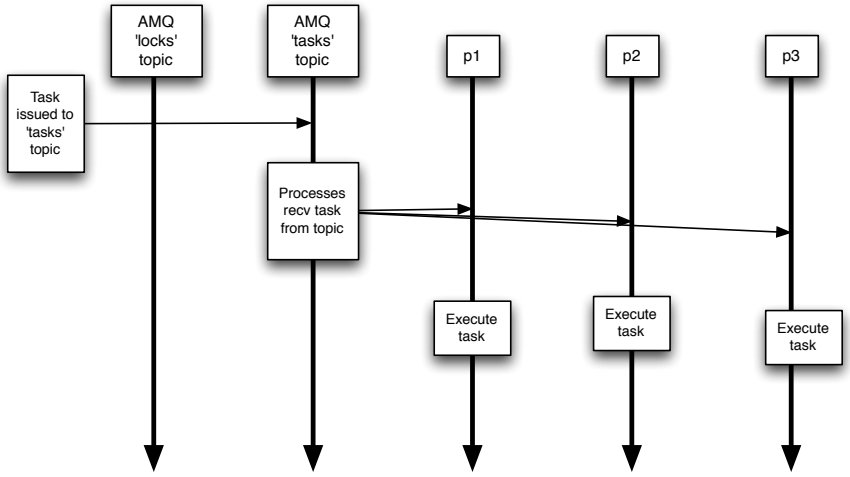
- On start up, a random ID is generated in the 1-1,000,000,000 range. This is to reduce the likelihood of an ID collision amongst 1000 processes 1 in a million. The nodes ID is used to identify it for lock granting and controller election.
- Task receipt and execution: provided by the **ClientListener** and **Task** classes.
- It sends and received lock requests, grants, denials, and unlocks via the **Lock** and **LockListener** classes.
- It manages electing a new controller via the **Election** and **ControllerListener** classes. The process with the highest ID in an election should win.
- The singleton class **ProcessConfig** stores runtime configuration data, such as ActiveMQ connection details.
- The singleton class **ProcessState** stored runtime state information about the system - such as the currently executing task, election state, and most importantly whether or not the process is a **controller**.
- A **LockTable** class, used by the controller system to establish who has a lock on which task.

On start up, ClientProcess sets up the asynchronous listeners on the three queues mentioned: 'tasks', 'locks' and 'controller', then enters an event loop based on the state within ProcessState - requesting new locks; handling lock timeouts, managing the election process.

Unless a task is received requiring a lock, no ClientProcess promotes itself to be the controller, since no controller is yet required.

Concurrent Task Execution

On submission of a task by TaskSubmitter, the ClientListener handler triggers and assigns the Task to the node. If this task is set to run concurrently (that is Task.require_lock is false), the following events take place:



In the above diagram, the three processes - p1, p2, p3 - receive the 'Task' object from the topic, and since no lock is required, they simply execute the task. No controller is required, and if one is not elected (due to failure of the existing controller, or recent system start up), no election will be called - as it is unnecessary.

Sequential Task Execution

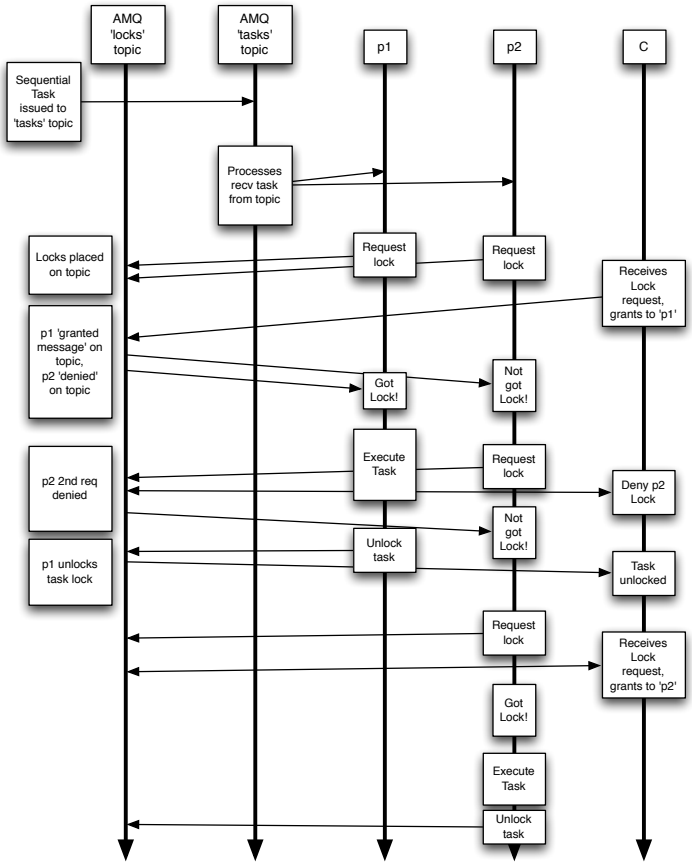
If a sequential task needs to be run (through submission of a Task object with Task.require_lock true), then the following event sequence occurs:

In the diagram opposite, we have controller 'C', and two processes 'p1', and 'p2'. 'C' is a role within p1 or p2, but is separated here for clarity.

p1 and p2 both receive a sequential task from the task topic, and around the same time submit a lock request to the 'lock' topic. The controller reads the first of these (from p1 in our example), and issues the lock granted message to the topic. The subsequent lock request from p2 is denied as a result, and a message is sent to the topic informing p2.

Whilst p1 is executing the task, p2 is in its main event loop, repeatedly requesting a lock grant from the controller. This is repeatedly denied.

Once p1 has completed its task, it releases the lock by submitting an 'unlock' message to the topic, which frees the lock on the controller. The next lock request from p2 then is granted, and execution/ unlock continues on that process.



Controller Election

In the event of controller failure or other unavailability, on receipt of a task requiring a lock, a **controller election** is requested by the first process to notice. This is performed by the implementation of a time-out on lock receipt - if a 'granted' or 'denial' message is not received by a process in a configurable amount of time (5s, by default), the controller is assumed to have failed by the process where the time-out occurred. It then announces an election in the following manner:

Here we have two processes, 'p1' and 'p2', and our ActiveMQ topics 'controller', 'locks', and 'tasks'. No process is acting as the controller, since a failure has occurred previously.

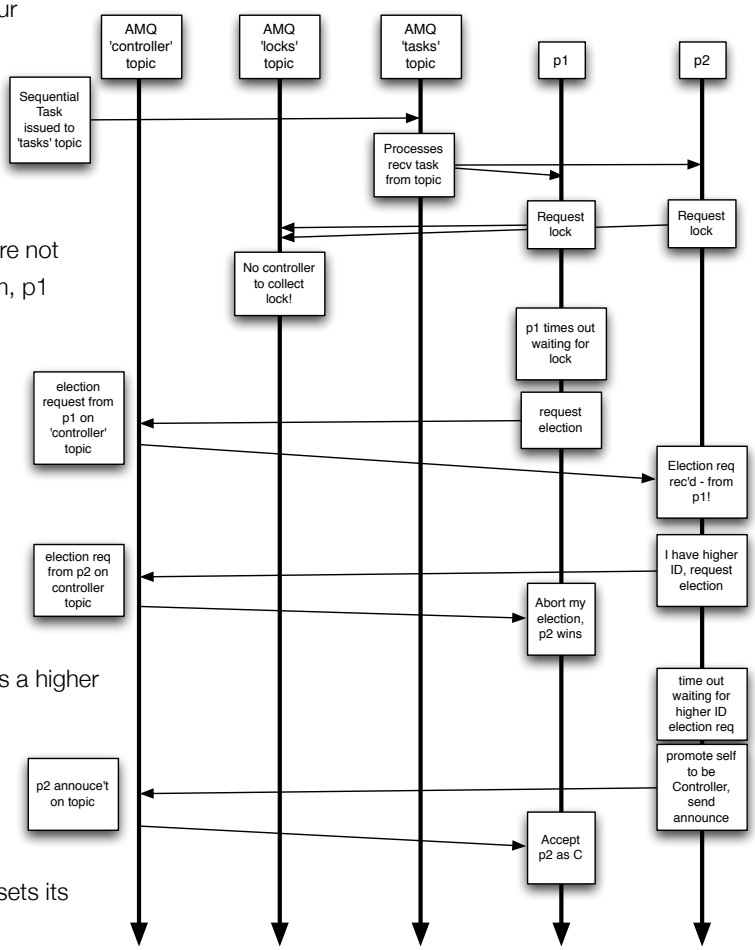
p1 has ID 1, p2 has ID 2.

Since the two lock requests issues by p1 and p2 are not answered - as there is no controller to answer them, p1 times out awaiting a response, and announces an election request message onto the 'controller' topic. This is received by ControllerListener on both p1 and p2. p1 ignores it, since it issued the request, but p2 recognises that the election request has come from an ID (1) lower than its own ID (2).

p2 then issues its own election request.

p1 receives this, and hence now knows that p2 has a higher ID, and so it should drop its own election request.

p2 however does not receive any further election requests, since there are no processes with higher ID that can submit one. After a short time-out period, it announced that it is a controller, and sets its state to act as the controller process.



Monitoring the cluster

As the system has ActiveMQ (or in fact, any other JMS broker) at its core, the state of the cluster can be interrogated by simply looking at the Admin console:

- The number of consumers highlights the number of processes in the cluster.
- The 'tasks' topic statistics highlight the business (and unprocessed) work list.
- The 'controller' topic statistics indicate if controller re-elections are taking place.
- The 'locks' topic indicate if lock requests are being submitted and processed.

Future Enhancements

Add a mechanism to selectively run a task on a subset of processes

For example, in the case of the task being ‘restart the Apache httpd process’, this may only need to be run on the ‘B’ cluster of nodes in a web farm. Other nodes may not even have an httpd process to restart. Some form of discovery process, based on known information about the environment of the process, could be created to ensure that the task is only scheduled for execution on these valid subsets.

Add a web interface to submit tasks and query the state of the cluster

The reason I selected JSON as the serialisation type for the Locks, Tasks and Election messages was so that a future system could be added in any programming language that supports talking to the JMS broker - either via JMS itself or a bridge protocol such as STOMP or AMQP. By keeping the message format simple and human parsable (as JSON provides), this is made trivial.

A web (or other query system) can then easily be written to submit ‘report state’ tasks to the processes in the cluster, and then collate the replies into a status page. It would be beneficial to have the ‘selective task’ operation in place before running this on large clusters of nodes however, to reduce return message traffic from processes.