Analysis on Thread Pool Module - Floodlight

CMPE: 207 (NETWORK PROGRAMMING AND APPLICATION)

Name: Sashank Malladi

SJSU ID: 010466651

Class Id: 34

1. Introduction

Software- Defined Networking (SDN) is an architecture decoupling network control and forwarding functions. It enables network control to become directly programmable and the underlying infrastructure to be abstracted for applications and network services.

An SDN controller is an application in software-defined networking (SDN) that manages flow control to enable intelligent networking. It can be considered as an operating system (OS) for a network. SDN controllers are based on protocols, such as OpenFlow, that allow servers to tell switches where to send packets. Any communications between applications and devices have to go through a controller.

Floodlight is one such SDN Controller offered by Big Switch Networks that works with the OpenFlow protocol to manage traffic flows in a software-defined networking (SDN) environment.

2. Floodlight Controller

Floodlight Controller is an Apache licensed Java-based OpenFlow controller. It is one of the significant contributions from Big Switch Networks to the open source community.

Floodlight support both northbound and southbound operations. When the controller starts running, all the corresponding configured modules will become active. The northbound REST APIs exposed by all the running modules becomes available through specified REST port. Any application can interact with the controller by sending http REST commands. On the other hand, at the southbound, the provider module of Floodlight will start listening on the OpenFlow-specified TCP-port for connections from the OpenFlow switches.

Floodlight, currently supports OpenFlow 1.0. Versions of 1.3 and 1.4 are in the pipeline. With an extensible Java development environment, and enterprise-grade core engine, Floodlight is both an easy to use and robust SDN controller.

3. Floodlight Architecture

Floodlight is considered to possess modular architecture because it provides various features and abstractions for controlling open flow networks. For optimum utilization of resources, Floodlight relies on multi-threading and can handle several million new flows per second.

The core architecture includes various modules, such as topology management, device/end-station management, path/route computation, infrastructure for web access (management), counter store (OpenFlow counters), and a state storage system, that are well stitched by module-management system.

In-built services defines the capability of the controller, and these services are the ones that are used by the northbound-applications. For utilities like storage, threads and web-UI, Floodlight includes services ranging from discovering network states and events to enable switch-communication.

Figure 1 at appendix depicts Floodlight Architecture with its modules.

4. Analysis on Concurrency

4.1 Thread Pool

Thread pool can be considered as group of pre-instantiated idle threads waiting to accept tasks. Considering scenario of large number of short tasks, thread pools are preferred instead of creating new thread for each task. Thread pools help in preventing overhead of creating and destroying threads large number of times. However, it is not necessary in scenario of small number of long running tasks.

Implementation of thread pools vary with the environment utilized. But the following containers are common and required in any implementation.

- APIs to create threads and maintain them in ideal state.
- A data structure to store created threads like queues with a provision to add a new threads or retrieve existing threads
- An abstract class or type for executing assigned tasks and properly synchronize threads in a pool. Concepts like mutex, semaphore or conditional variables can be utilized for synchronization.

When a thread pool is created, they can be instantiated directly with certain number of threads or new threads can be added based on need in implementation.

When the pool is assigned with a Task, it retrieves a thread from the container (or waits for one to become available if the container is empty), assign it with given task, and start executing it. Invoking an *execute*() method of the Task assigned makes ideal thread to resume execution. Once execution is complete, the thread allocate itself back to the pool group for re-use and then meets its barrier, putting itself to sleep until the cycle repeats.

4.2 Concurrency in Java

As floodlight controller is built on Java platform, it is a good option to observe provisions of concurrency included in java and then relate it to thread pool module in floodlight controller.

In Java version 5, java.util.concurrent package with set of classes that helps in developing concurrent applications was introduced. A specific interface found inside mentioned package (*java.util.concurrent.ExecutorService*) represents an asynchronous execution mechanism capable of executing tasks in background.

ExecutorService can be considered as interface inside Java to create a thread pool. Following code provides an example of ExecutorService usage.

```
ExecutorService executorService = Executors.newFixedThreadPool(10);

executorService.execute(new Runnable() {
public void run() {
System.out.println("Asynchronous task");
}
});

executorService.shutdown();
```

A new ExecutorService instance is created using *newFixedThreadPool()* factory method. As we are passing 10 as a parameter to it in the above code, it creates a thread pool with 10 threads inside it.

Runnable interface is the passed to the *execute*() method. This cause Runnable code to execute by one of the threads maintained by thread pool created in above step.

Executor service being an interface, it needs implementation. So, there are two implementations offered under Executor service interface in java.util.concurrent package.

- ThreadPoolExecutor
- ScheduledThreadPoolExecutor

The *threadPoolExecutor* can contain varying amount of threads. The number of threads inside created pool can be determined using *corePoolSize* and *maximumPoolSize* variables. If number of threads created are less than corePoolSize, a new thread is created inside pool when a new task is delegated. If queue of tasks is full but number of treads running are less than maximumPoolSize, again in this scenario also a new thread will be created to serve the task.

The following code represents constructor invoked when a new *threadPoolExecutor* instance is created.

```
int  corePoolSize  =    5;
int  maxPoolSize   =   10;
long keepAliveTime = 5000;

ExecutorService threadPoolExecutor =
new ThreadPoolExecutor(
corePoolSize,
maxPoolSize,
keepAliveTime,
TimeUnit.MILLISECONDS,
new LinkedBlockingQueue<Runnable>()
);
```

The *java.util.concurrent.ScheduledExecutorService* is one of the other *ExecutorService* implementation in which user has a provision to schedule tasks. User may opt for an execution after a delay or execute repeatedly with a fixed time separation between each execution. It is important to observe that tasks are executed by worker threads but not by the thread which is used for assigning task to *ScheduledExecutorService*.

Below is an example code for implementation of *ScheduledExecutorService*.

```
ScheduledExecutorService scheduledExecutorService =
Executors.newScheduledThreadPool(5);

ScheduledFuture scheduledFuture =
scheduledExecutorService.schedule(new Callable() {
public Object call() throws Exception {
System.out.println("Executed!");
return "Called!";
}
},
5,
TimeUnit.SECONDS);
```

It can be observed that 5 threads are created in a pool using *newScheduledThreadPool* method. And then, a task is passed to the thread pool using schedule method. The last two parameters in schedule method indicates that assigned task should be executed after 5 seconds of delay.

5. Analysis on Thread Pool Module

5.1 Description

ThreadPool is one of the module in Floodlight controller providing wrapper for Java's *ScheduledExecutorService* described in above section. It provides multithreading across different modules in the controller. And these threads will be running at specific times or periodically because they were created based on *ScheduledExecutorService* implementation.

5.2 Services Provided

All the services from this module are provided using *IThreadPoolService* interface.

5.3 Directory Path to the Module

This module consists of two files an interface (IThreadPoolService.java) and a corresponding implementation (ThreadPool.java). These files can be found at location specified as follows ***"floodlight/src/main/java/net/floodlightcontroller/threadpool/***".

5.4 Analysis on IThreadPoolService Interface

It is a simple interface abstracting implementation of thread pool concept utilized in floodlight controller. Below are important lines of code found in this interface file.

```
package net.floodlightcontroller.threadpool;
import java.util.concurrent.ScheduledExecutorService;
import net.floodlightcontroller.core.module.IFloodlightService;

public interface IThreadPoolService extends IFloodlightService {

public ScheduledExecutorService getScheduledExecutor();

}
```

Initially, a new package is defined inside floodlight controller to form thread Pool module using package key word.

As Described earlier, the thread pool module inherits base Java util to support its own implementation. So in this second line it inherits "*java.util.concurrent.ScheduledExecutorService*" using import keyword.

As this module is an extension to services provided by the floodlight controller, this interface extends base interface of floodlight controller by importing "*net.floodlightcontroller.core.module.IFloodlightService***"** using import key word and provide extension using extends keyword.

This interface provides abstract for a method *getScheduledExecutor***.** This method when referenced will return underlying ScheduledExecutorService for native access. It throws an exception by the state "*IllegalStateException* " when there no *ThreadPoolTaskScheduler* initialize yet.

5.5 Analysis on ThreadPool Class

Implementation for the interface *IThreadPoolService* is provided in ThreadPool class. This file starts with importing all the Java utils required in providing floodlight controller concurrency using threads concept. Some of the imported libraries are as follows.

- java.util.concurrent.Executors
- java.util.concurrent.ScheduledExecutorService

- java.util.concurrent.ThreadFactory
- java.util.concurrent.atomic.AtomicInteger

Executor's class involves in creation of an object that executes submitted Runnable tasks. It decouples task submission from the actual execution of the task. An Executor is normally used instead of explicitly creating threads. For example, instead of invoking new Thread (new (RunnableTask ())).start () for each of a set of tasks, we might directly use:

```
Executor executor = anExecutor;
executor.execute(new RunnableTask1());
executor.execute(new RunnableTask2());
```

ScheduledExecutorService is one of the implementations for ExecutorService to schedule tasks.

The *ThreadFactory* is utilized for dynamic creation of threads on demand. It removes hardcoding in creation of new threads. Sample code utilizing thread factory is as follows

```
class SimpleThreadFactory implements ThreadFactory {
public Thread newThread(Runnable r) {
return new Thread(r);
}
}
```

Atomic integers are used for proper synchronization between different threads.

In the public class ThreadPool we can observe that several base methods were overridden to match functionalities in supporting floodlight controller. Following are the few important functions overridden in this class

- getScheduledExecutor
- getModuleServices
- getServiceImpls
- init

getScheduledExecutor method returns current instance of *ScheduledExecutorService* for native access to the object that invoked it.

When the floodlight controller starts, it loads all the modules configured and initialize them for successful execution. So the other two methods overridden *getModuleServices* and *getServiceImpls* will be called by *loadModulesFromList* Method in *floodlightModuleLoder.java* for successful loading of thread pool Module.

GetModuleServices adds *IThreadPoolService* class on to the existing list of modules required to load in this instance of the controller. Array list data type is used to maintain list of services.

```
@Override
public Collection<Class<? extends IFloodlightService>>
getModuleServices() {
Collection<Class<? extends IFloodlightService>> l =
new ArrayList<Class<? extends IFloodlightService>>();
l.add(IThreadPoolService.class);
```

```
return l;
}
```

GetServiceImpls method adds implementations to the list of services loaded in the current instance of the controller. So this maps *ThreadPool* class to the *IThreadPoolService* loaded during GetModuleServices function call. Hash Map data type is used in mapping services to their corresponding implementations.

```
@Override
public Map<Class<? extends IFloodlightService>, IFloodlightService>
getServiceImpls() {

Map<Class<? extends IFloodlightService>,
IFloodlightService> m = new HashMap<Class<? extends      IFloodlightService>,
IFloodlightService>();
m.put(IThreadPoolService.class, this);
return m;
}
```

When loading of dependencies of all modules are completed, then each module is initialized by the controller by making a function call (initModules). So there exists an **init** method inside thread pool implementation to initialize this module. Flowing is the code for initialization.

```
@Override
public void init(FloodlightModuleContext context)
throws FloodlightModuleException {
final ThreadGroup tg = new ThreadGroup("Scheduled Task Threads");
ThreadFactory f = new ThreadFactory() {
AtomicInteger id = new AtomicInteger();

@Override
public Thread newThread(Runnable runnable) {
return new Thread(tg, runnable,
"Scheduled-" + id.getAndIncrement());
}
};
executor = Executors.newScheduledThreadPool(5, f);
}
```

From the above code it is possible to observe that an executor is created for this instance while initialization. *newScheduledThreadPool* accepts two parameters corePoolSize and ThreadFactory. As corePoolSize is 5, this module maintain minimum 5 threads though they are idle. A corresponding thread factory is also defined for thread creation.

6. ThreadPool Utilization

Thread pool module is utilized by several other modules in the controller to improve their efficiency. Following are few modules utilizing multithreads for their implementation

- Sync Manager
- DHCP Server
- Link Discovery Manager

Let us consider each module and observe their utilization of threads.

6.1 Multithreading in SyncManager

The sync service provides a high-performance in-memory database for fault and partition-tolerant replication of state data. It provides eventually consistent semantics with versioning through vector clocks and allows custom handling of resolution of inconsistent writes.

SyncManager.java utilizes multithreading in its startup function. It utilizes different threads for following implementations.

6.1.1 Cleanup Task

SyncManager creates a singleton task to perform periodic maintenance/cleanup on local storage. The following code shows how this task is assigned to a thread pool.

```
cleanupTask = new SingletonTask(threadPool.getScheduledExecutor(),
new CleanupTask());
cleanupTask.reschedule(CLEANUP_INTERVAL +
random.nextInt(30), TimeUnit.SECONDS);
```

From the above code it can be observed that a task is assigned to an existing thread pool. *ThreadPool.getScheduledExecutor* gets current instance of a thread pool created and then it is assigned with cleanup task using new *singletonTask* instance. *Reschedule* method specifies interval with which this task should be repeatedly execute.

6.1.2 Anti-entropy Task

SyncManager creates a singleton task to perform periodic anti entropy between nodes connected. The following code shows how this task is assigned to a thread pool.

```
antientropyTask = new SingletonTask(threadPool.getScheduledExecutor(),
new AntientropyTask());
antientropyTask.reschedule(ANTIENTROPY_INTERVAL +
random.nextInt(30), TimeUnit.SECONDS);
```

Similar to Cleanup task, in the above code, anti-entropy Task is assigned to current instance of the thread pool.

6.1.3 Update Configuration Task

SyncManager also creates a singleton task using following code to perform periodic rescan on node configurations similar to cleanup and anti-entropy tasks.

```
updateConfigTask =new SingletonTask(threadPool.getScheduledExecutor(),
```

```
new UpdateConfigTask());
updateConfigTask.reschedule(CONFIG_RESCAN_INTERVAL, TimeUnit.SECONDS);
}
```

6.1.4 Maintain Queue task

SyncManager creates 2 Worker threads in a thread pool to drain the queue of writes that need to be sent to the all connected nodes. This ensures proper utilization of space during I/O operation. In this case SyncManager does not utilize existing thread pool but creates a new thread group named “Hint Workers”.

```
final ThreadGroup tg = new ThreadGroup("Hint Workers");
tg.setMaxPriority(Thread.NORM_PRIORITY - 2);
ThreadFactory f = new ThreadFactory() {
AtomicInteger id = new AtomicInteger();

@Override
public Thread newThread(Runnable runnable) {
return new Thread(tg, runnable,
"HintWorker-" + id.getAndIncrement());
}
};
hintThreadPool = Executors.newCachedThreadPool(f);
for (int i = 0; i < SYNC_WORKER_POOL; i++) {
hintThreadPool.execute(new HintWorker());
}
```

It can be observed that a thread factory is utilized for creation of new threads. Execution method is used for assigning tasks to threads in Hint Workers thread pool over a loop.

6.2 Multithreading in DHCP Server

There is an implementation of DHCP server module for nodes connected through floodlight controller. DHCP generally assign IP addresses to all nodes with a specified lease time. It requires a mechanism to collect all the expired IP addresses and add them back to address pool for reallocation. So for this very own purpose, DHCP module in floodlight controller utilizes a background task running in separate thread.

The utilization of the background thread can be observed in the following code.

```
private static ScheduledThreadPoolExecutor leasePoliceDispatcher;

// Monitor bindings for expired leases and clean them up
leasePoliceDispatcher = new ScheduledThreadPoolExecutor(1);
leasePolicePatrol = new DHCPLeasePolice();
/*leasePoliceOfficer = */
leasePoliceDispatcher.scheduleAtFixedRate(leasePolicePatrol, 10,
DHCP_SERVER_LEASE_POLICE_PATROL_PERIOD_SECONDS, TimeUnit.SECONDS);
```

In the above code, a new instance for existing thread pool *ScheduledThreadPoolExecutor* is created with one worker thread. And this thread is assigned with a task that should be executing

repeatedly with a gap of 10 seconds. So now the DHCP server checks for expired IP leases every 10 second to add them back to address pool. This implementation can be observed from DHCP commands on the console of the eclipse. Attached *figure 2* at appendix depicts this scenario.

6.3 Multithreading in Link Discovery Manager

In traditional networks, Spanning Tree or routing protocols are often used in topology management. As network grows in size managing topologies become more complex. In contrast, Now SDN controllers have a central view of all network components and can therefore greatly simplify topology management.

To let this happen, Floodlight implements a sophisticated mechanism for automatically detecting the topology of an OpenFlow network. Using a link-discovery module, the controller generates both LLDP and broadcast packets (referred to as BDDPs) and sends them to all neighboring switches on a regular basis. Assuming all switches consume LLDP messages and forward broadcast packets, Floodlight can identify active connections by receiving its own messages and computing the network topology.

So, Link Discovery manager utilizes background threads to send LLDP or BDDP packets. The following code shows how multithreading is utilized in this module. Initially, it get a current instance of the existing thread pool using

```
ScheduledExecutorService ses = threadPoolService.getScheduledExecutor();
```

It then creates a singleton task by assigning a runnable code to the thread pool for sending LLDP packets. It can be observed in following code snippet.

```
// To be started by the first switch connection
discoveryTask = new SingletonTask(ses, new Runnable() {
@Override
public void run() {
try {
discoverLinks();
}
catch (StorageException e) {
shutdownService.terminate("Storage exception in LLDP send timer. Terminating
process " + e, 0);
}
catch (Exception e) {
log.error("Exception in LLDP send timer.", e);
}
finally {

if (!shuttingDown) {
// null role implies HA mode is not enabled.
if (role == null || role == HARole.ACTIVE) {
log.trace("Rescheduling discovery task as role = {}",
role);
discoveryTask.reschedule(DISCOVERY_TASK_INTERVAL,
TimeUnit.SECONDS);
}
else {
log.trace("Stopped LLDP rescheduling due to role = {}.",
role);
}
}
```

```
}
}
});
```

And then rescheduling is done using reschedule function

```
discoveryTask.reschedule(DISCOVERY_TASK_INTERVAL,
TimeUnit.SECONDS);
```

In the same manner thread pool is utilized for sending BDDP packets at regular interval of time when switch port tuples are detected.

Attached *figure 2* at appendix depicts Link Discovery Manager sending LLDP packets.

7. Conclusion

So the wrapper provided by the floodlight to the Java concurrency package in Thread Pool module is observed in detail. Analysis on Different modules utilizing thread pool is done considering factors like performance improvement and proper synchronization.

8. Acknowledgment

I would like to thank Dr. Younghee Park for introducing and instructing various steps involved in analysis of this Floodlight SDN controller. Also, I would like to thank her group of research assistants for meaningful discussions on functioning of the controller.

9. References

- https://www.sdxcentral.com/resources/sdn/sdn-controllers/open-source-sdn-controllers/what-is-floodlight-controller/
- http://thenewstack.io/sdn-series-part-v-floodlight/
- http://mininet.org/overview/
- http://tutorials.jenkov.com/java-util-concurrent/index.html
- http://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/scheduling/concurrent/ThreadPoolTaskScheduler.html
- https://github.com/floodlight/floodlight

10. Appendix

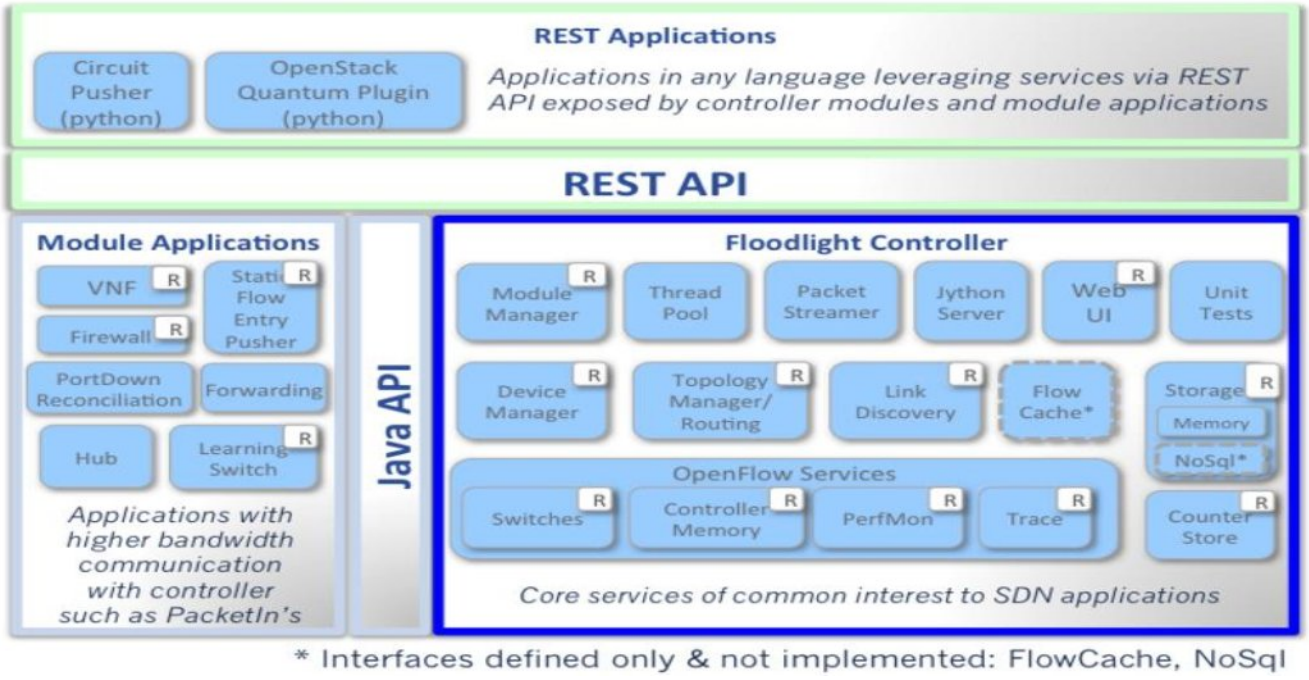

Figure1

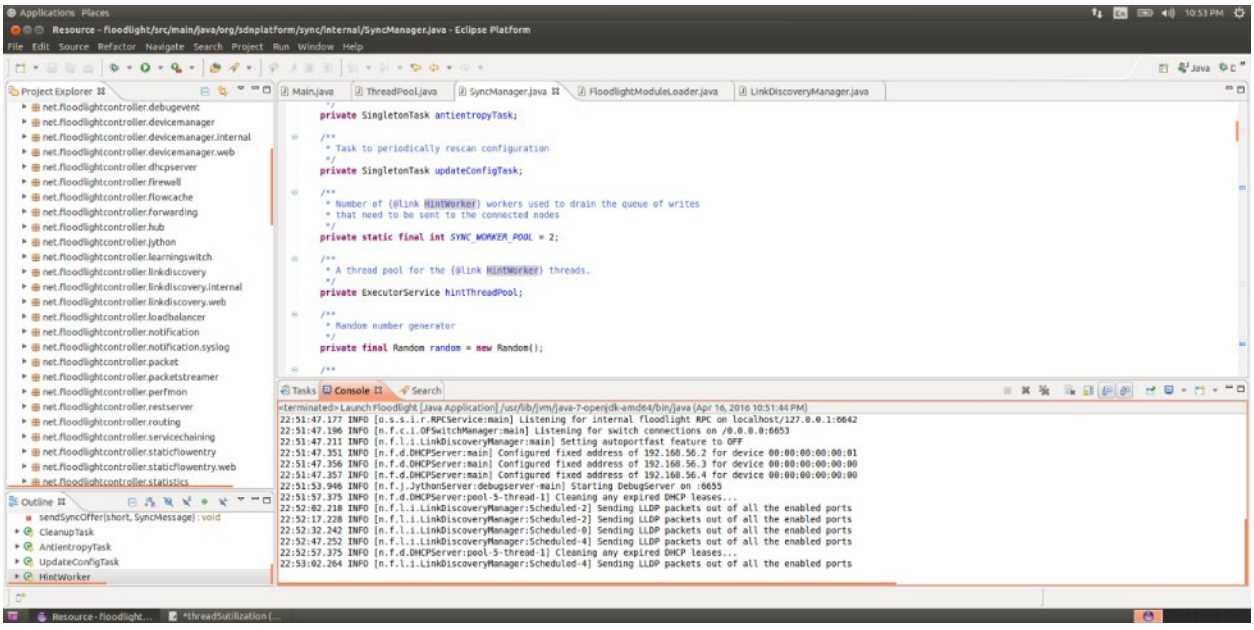

Figure 2