A Java Library for Conditional Request Serialization of Asynchronous Requests in Parallel

Arun Yadav
Samsung Research Institute - Bangalore
arun.y@samsung.com

Abstract

This paper describes the design and implementation of a Java library for serializing asynchronous requests by some request criteria, yet parallely processing requests which are independent (or not having same request criteria). This is generally required in scenarios where requests are arriving asynchronously and can be processed parrallely with a condition that request with certain criteria (e.g. request userId) to be processed in chronological order. The measured performance shows good improvement over single threaded processing.

1 Introduction

Most of the time we use queue based communication if we wanted asynchronous request processing. However cases where messages are being received in asynchronous manner yet we want to serialize/order/or maintain happen-before relationship of the request processing by some criteria say, all requests with same userid to be processed in serial order, we can't simply use queue, because the queue listeners/consumers (if more than one on a single queue) will pick requests independent of other requets and processing will happen parallel. In such situation the happen before relation among requests under same key will be lost. This paper discuss the design and implementation of a Java library that addreses above use case.

2 Design

Asynchronous requests can easily be processed in parallel given there is no inter-dependency among arriving request. A simple work queue with multiple listeners (consumers) design can be utilized. Next free listener can pick-up the request and start execution in parallel however, in scenarios where processing needs to maintain happen-before relationship among set of arriving requests with same request key, we need a controlled excution of request. The design proposed here takes care of serializing set of requests where happen-before relation is to be maintained, yet parallelly executing other requests where happen-before relatation need not be maintained. The design also takes care of not engulfing the system (by not creating too many processing thread) when number of requests arrives at rate greater than rate at which application can process requests. The library can scale veritcally by increasing the number of processing cores of a given machine.

Below are core desing elements

- A Worker Thread **(WT)** with internal/local in-memory work queue.
- A fixed size Worker Thread Pool **(WTP)** of **WTs** is maintained while system is loaded. The default behaviour is to create as many **WTs** as available CPUs to JVM.

- A **WTP** will also maintains a Request Key **(RK)** to **WT** mapping **(RK-WT-M)** for allocating and de-allocating **WT** against **RK**.
- A Single Listener Thread **(SLT)** to receive request and submitting the same to allocated **WT**

The above four elements co-ordinate among themshelf to meet the system requirement in a following way.

1. A **SLT** upon receiving a request **(R)** along with associated **RK**, gets a **WT** from **WTP**. This call blocks if all **WT** in **WTP** is already assigned to some **RK**.
2. Upon receiving **WT**, **SLT** attempts to assign this **R** to received **WT**.
3. **SLT** maintains an attempt count and attempt delay in case assignments fails in previous step, to re-assign **R** by getting new **WT** from **WTP**. The assignment can fail in case **WT** deactivated iteself (de-allocated against mapped **RK**) after it was given to **SLT** by **WTP**.
4. Upon receiving assignment request from **SLT**, the **WT** first gets a locks for local work queue and its state variable. Upon receiving lock, **WT** checks if state is still active (mapped with **RK**), if yes it adds request into local work queue and release the lock and return to **SLT** with success. If given **WT** is not active, it returns failure to **SLT**, upon which **SLT** may re-attempt as described in the step above.
5. **WT** when active (i.e. allocated/mapped against some **RK**) is always in two states:
   (a) Either blocking (with timeout) on local work queue for new request
   (b) Or processing the request
6. While in blocking state, **WT** has configurable timeout, which gives an opportunity to de-allocat itself when no new request is being assigned before timeout since completion of last processing.
7. Upon timing-out while blocking on work queue, **WT** aquires a lock for local work queue and its state variable. Test again the local work queue to check if something got added after timing out and aquiring lock.
8. If something is found, then **WT** goes ahead and process the received request. If not, then **WT** prepare itself to de-allocated itself aganst mapped **RK** and release itself to **WTP**. This step is essential for optimal performance of this system. During de-allocation, the **WT** can't accept new assignment by **SLT** (**SLT** will block during that time). And **WT** sets its state as inactive. Upon de-allocation/releaseing itself to **WTP**, **WT** goes in waiting state (thread.wait()). After which its the responsibilty of **WTP** to notify(thread.notify()) this **WT** while re-allocating to some **RK** and giving out this **WT** to **SLT** in step 1.
9. Because the same **WT** can be re-used for processing **R** with other **RK**s. The more time **WT** blocks itself against new **R** in local work queue, the less efficient the overall system becomes. However quickly timing-out and de-allocating itself can also have adverse effect of re-allocation and reloading the initial context for processing incase request processing has heavy initiallization cost per **RK**. Client application can however configure this parameter based on average rate of request arrival.
10. The **WTP** has two functions:
    (a) Returning free **WT** from pool and mapping it against **RK**
    (b) Releasing **WT** into pool and un-mapping it against **RK**

11. While returning free **WT** from pool, **WTP** aquires a lock for worker thread-request-key-map **(RK-WT-M)**

12. Checks if some **WT** is already mapped against given **RK**. If found return the same and release the lock for **RK-WT-M**

13. If not found borrow a new **WT** from pool. If borrowing succeeds map the **RK** with **WT** in **RK-WT-M**.

14. If borrowing fails, it means pool is exhausted and **WTP** goes on waiting (thread.wait()) till some WT is released back into pool.

15. When **WT** determines that there is no new pending request, it calls **WTP** release() function to release itself. The **WTP** release function again aquires a lock for **RT-WT-M** and deletes its entry against **RK** and then releases to pool notifing (thread.notify()) to wake up **SLT** thread waiting on availability for new **WT** in **WTP** allocate function described above.

3 Implementation

The framework is built on top of `java.concurrent.util` and `org.apache.commons.pool` packages. The singleton `WorkerThreadPool` object (referred as `WTP` above) maintains a pre-defined (configurable) pool of
texttttWorkerThread objects, I have used `org.apache.commons.pool.impl.GenericObjectPool` to maintain the same. The `AsyncRequestSerializer` is an entry point into framework, which accepts an instance of `Work` insterface along with a request key. It returns an instance of `java.util.concurrent.Future` which client can use to retrieve response of submitted work request. A Java `java.util.HashMap` is being used to keep the map of **RK** to **WT**. The access to this map is synchronized between thread requesting a **WT** vs. thread returning the **WT** back into the pool.

Below is the code fragement of `WorkerThreadPool`

```
public class PoolableWorkerThreadPool<U> {



  private final GenericObjectPool<PoolableWorkerThread<U>> workerThreadPool;

  private final Map<String, PoolableWorkerThread<U>> requestKeyWorkerThreadMap;

  private final Object workerThreadPoolLock = new Object();


  public PoolableWorkerThreadPool(

        final AsyncRequestSerializerConfig asyncRequestSerializerConfig) {

     int availableProcessor = Runtime.getRuntime().availableProcessors();

     Config config = new Config();

     int poolsize = Integer.parseInt(asyncRequestSerializerConfig.workerThreadPoolSize);

     config.maxActive = poolsize <= 0 ? availableProcessor

           : poolsize;

     config.whenExhaustedAction = GenericObjectPool.WHEN_EXHAUSTED_FAIL;
```

```
    this.workerThreadPool = new GenericObjectPool<PoolableWorkerThread<U>>(
            new PoolableWorkerThreadFactory<U>(this,
                    asyncRequestSerializerConfig), config);
    this.requestKeyWorkerThreadMap = new HashMap<String, PoolableWorkerThread<U>>();
}

public PoolableWorkerThread<U> getPoolableWorkerThread(final String requestKey)
        throws Exception {
    long st = System.currentTimeMillis();
    synchronized (workerThreadPoolLock) {
        PoolableWorkerThread<U> poolableWorkerThread = requestKeyWorkerThreadMap
                .get(requestKey);
        if (poolableWorkerThread == null) {
            do {
                try {
                    poolableWorkerThread = workerThreadPool.borrowObject();
                } catch (NoSuchElementException e) {
                    try {
                        workerThreadPoolLock.wait();
                    } catch (InterruptedException ie) {
                        continue;
                    }
                }
            } while (poolableWorkerThread == null);
            requestKeyWorkerThreadMap.put(requestKey, poolableWorkerThread);
            poolableWorkerThread.setCurrentRequestKey(requestKey);
        }
        return poolableWorkerThread;
    }
}

public void returnPoolableWorkerThread(PoolableWorkerThread<U> workerThread)
        throws Exception {
    long st = System.currentTimeMillis();
    synchronized (workerThreadPoolLock) {
        requestKeyWorkerThreadMap.remove(workerThread
                .getCurrentRequestKey());
```

```
            workerThreadPool.returnObject(workerThread);
            workerThreadPoolLock.notifyAll();
        }
    }
}
```

The `WorkerThread` class once assigned to a **RK** starts processing incoming `Work` on a single thread, the single thread allows the system to process work request in serial order per request key. The incoming `Work` will be pushed into a `java.util.BlockingQueue` upon which the `WorkerThread` blocks till timeout. The BlockingQueue implementation takes care of blocking the **WT** till some new `Work` is arrived or timeout occurs. In current implementation the timeout is pre-defined configurable which should be assigned value based on general nature of rate of new Work per request key. Given too large value will starve other request key and give too small value may not give the benfit of localization during processing of a given request key. In future we can improve the design to dynamically choose the timeout value by observing the previous rate of arrival for a given request key. The idea of local work queue inside `WorkerThread` is similar to that of "Actor" model".

The `WorkerThread` maintains an internal boolean state `isActive` to indicate if its assigned to some request key or not. As soon as `WorkerThread` is returned from a pool, `isActive` will be marked as true. Till the state isActive is set, the `WorkerThread` will accept new `Work` into its local work queue.

Below is the complete code for `WorkerThread`

```
public class PoolableWorkerThread<U> extends Thread {

    private final PoolableWorkerThreadPool myPool;
    private final AsyncRequestSerializerConfig asyncRequestSerializerConfig;

    private ExecutorService executorService = Executors
            .newSingleThreadExecutor();

    private final BlockingQueue<Future<U>> localRequestQueue = new LinkedBlockingQueue<Future<U>>();
    private final Object localRequestQueueLock = new Object();
    private boolean isActive = false;

    public PoolableWorkerThread(final PoolableWorkerThreadPool myPool, final AsyncRequestSerializerConfi
        this.myPool = myPool;
        this.asyncRequestSerializerConfig = asyncRequestSerializerConfig;
    }

    @Override
    public void run() {
        while (true) {
```

```java
      try {
        Future<U> request = localRequestQueue.poll(
            Integer.parseInt(asyncRequestSerializerConfig.localRequestQueueTimeOut),
            TimeUnit.MILLISECONDS);
        if (request == null) {
          synchronized (localRequestQueueLock) {
            request = localRequestQueue.poll();
            if (request == null) {
              myPool.returnPoolableWorkerThread(this);
              isActive = false;
              currentRequestKey = null;
              do {
                try {
                  /*To awaken, please call activate()*/
                  localRequestQueueLock.wait();
                  break;//break from wait
                } catch (InterruptedException e) {
                  //"Interrupted or spurious wake up,
                  // will check if isActive is set"
                }
              } while (!isActive);
              continue;//continue to look for new work
            }
          }
        }
        /* make sure this executor service is singleThreadExecutor */
        long st = System.currentTimeMillis();
        request.get();
      } catch (Exception e) {
        //"Error while executing local requests"
      }
    }
    executorService.shutdown();
  }

  public Future<U> assign(Work<U> request) {
    synchronized (localRequestQueueLock) {
```

```
        if (isActive) {
          Future<U> future = executorService.submit(request);
          localRequestQueue.add(future);
          return future;
        } else {
          return null;
        }
      }
    }

    public void activate() {
      synchronized (localRequestQueueLock) {
        localRequestQueueLock.notify();
        isActive = true;
      }
    }
}
```

---

4 Conclusion

The current design guarantees serial processing of all request under same request key, yet parallely processing other request key. In our system where we have used this framework has improved the application throughput vs. serially processing requests across request key. The current design may however suffers from a starvation problem in scenarios where **RKs** are arriving at rate higher than the timeout set for **WT** and too many such request keys arrives at same time, not allowing mapped **WT** to release itself. This will cause stravation to other less frequent **RK**. Though we definately need to address this scenario, the present design assumed that no selected **RK** will engulf the system at much higer rate then rest. And we can keep the timeout small enough so that **WT** releases itself frequently. As future improvement we can include some ability to yield **WT** if its being mapped for long time by parking the pending requests in local work queue. In addition to incremental improvents, future work on this framework may include adding ability to achieve the same abilty on a distributed enviroment, that will allow horizontal scaling within the framework without externally controlling user synchronization across machine.

5 References

References

[1] Doug Lea, "A Java Fork/Join Framework" *Proceedings of the ACM 2000 Conference on Java Grande*, 2000.

[2] Rajesh K. Karmani, Gul Agah, "Actor" *http://www.cs.ucla.edu/ palsberg/course/cs239/papers/karmani-agha.pdf.*