CSE535: Asynchronous Systems

Phase - 3

Gandharv Kapoor 110898207
Sachin Tiwari 110924177

==

Classes

```
- class request{

            set objects {sub, res}
            int client_id
            int id
            long ts
            bool isWriteReq
            list[2] cachedUpdates
            list[2] readAttr
            list[] updates
            obj updatedObj
            obj rdonlydObj
            bool decision
}

- class version{

           long rts
           long wts
           list[] pendingMightRead
}
```

==

Data Structures maintained at each Coordinator

```
  #we maintain a version table map at each coordinator level which basically maintains a list
  #of versions(class defined above) for each (obj, attr) set as key
- versionTable
  map<(obj, attr), list<version>>

  #we maintain a readWaitingQueue which contains all incoming potentially-conflicting read-only
  #requests, and list of conflicting write requests. These read only requests are not added to
  #pendingMightRead till write requests are resolved.
- readWaitingQueue
  map<r_req, List<w_reqIds..>>
  #we maintain a writeWaitingQueue which contains all write requests and list of conflicting
  #ongoing read only requests. The write requests stays in this queue till ongoing read requests
  #to get resolved
- writeWaitingQueue
map<w_req, List<r_reqIds..>>

  #we maintain a attribute cache which for each object contains list of attribute value and
  #timestamps for each attribute type.
- cache
  map<obj, map<attr, list<{value, timestamp}>>>

========================================================
Policy Dependent Methods

- mightWriteObj(req) is an upper bound on the set of objects updated by req.
  mightWriteObj(req) subseteq {req.sub, req.res}.

- defReadAttr(x,req) is a set of attributes of x definitely read by req.

- mightReadAttr(x,req) is an upper bound on the set of attributes of x that might be read by req
  and are not definitely read by req (i.e., are not in defReadAttr(x,req)).

- mightWriteAttr(x,req) is an upper bound on the set of attributes of x that might be updated by req.

========================================================
1. Client()

        # check if its a write request with 1 might write obj
        if |mightWriteObj(req)| == 1:
                #send the read object first to coordR to minimize non-local messages
                req.isWriteReq = true
                Sends req to coord(obj id (1 or 2) in defReadAttr(x,req) union mightReadAttr(x,req))

        #although below case shouldnt come up i have tried to still handle it
        else if |mightWriteObj(req)| > 1:
                req.isWriteReq = true
                #send to any coordinator since both are write obj
                Sends req to coord(rand(1...2))

        #if not a write request
        else:
                req.isWriteReq = false
                Sends req to coord(obj(req,1))

        Wait until Receives decision d or timeout = t
                On timeout: #in case of some unknown failure
                        Resend req
        On decision:
            Return d


=====================================================
2. Coordinator

    On receiving req as coord(obj(req, 1)):

        x = obj(req, 1)
        req.ts = now()

        if req.isWriteReq:
            #there are no definite reads, because the request might abort
            for attr in defReadAttr(x,req) union mightReadAttr(x,req)    :
                latestVersionBefore(x,attr,req.ts).pendingMightRead.add(req.id)
        else:

            # traverse in coordinator level write waiting queue and check if their are any
            #conflicting attrs
            for w_req in writeWaitingQueue:
                #if attr found which are going to be updated by this write req soon
                if intersection of all attr in (mightReadAttr<x, req>) and (w_req.updates)
                is non-empty:
                    # add this write_req id in the dependency list of read_req
                    if readWaitingQueue.contains(req):
                        readWaitingQueue.get(req).addInList(w_req)
                    else
                    # if list not present create the list and add write_req id
                        readWaitingQueue.put(req, List<wreq>)
                    #return since we can't move ahead for this read_req now,
                    #this req will be soon waked up by last exiting write_req
                return

            # we know definite reads with their respective read timestamps
            for attr in defReadAttr(x,req):
                latestVersionBefore(x,attr,req.ts).rts = req.ts
            # add might be read attributes to pending list with respective read timestamps
            for attr in mightReadAttr(x,req):
                latestVersionBefore(x,attr,req.ts).pendingMightRead.add(req.id).


        req.cachedUpdates[1] = cachedUpdates(x,req)

        #to preventing unecessary extra communication, we check if this coordinator is
        #responsible for obj(req, 2)
        if coord(obj(req, 2)) ==  self.id:
            goto 2 and process obj(req, 2)
```

```
                else:
                        send req to coord(obj(req,2))

        On receiving <"restart", req> as coord(obj(req, 1)):

                #we know definitely it is a write request in case of restart tag

                #reset params of received request
                newReq = new request()
                newReq.objects = req.objects
                newReq.client_id = req.client_id
                newReq.isWriteReq = true
                req = newReq

                for attr in defReadAttr(x,req) union mightReadAttr(x,req):
                        v = latestVersionBefore(x,attr,req.ts)
                        v.pendingMightRead.remove(req.id)


                x = obj(req, 1)
                req.ts = now()

                for attr in defReadAttr(x,req) union mightReadAttr(x,req)    :
                        latestVersionBefore(x,attr,req.ts).pendingMightRead.add(req.id)

                req.cachedUpdates[1] = cachedUpdates(x,req)

                #to preventing unecessary extra communication, we check if this coordinator is
                #responsible for obj(req, 2)
                if coord(obj(req, 2)) ==  self.id:
                        goto 2 and process obj(req, 2)
                else:
                        send req to coord(obj(req,2))

=======================================================
3. Coordinator

        On receiving req as coord(obj(req, 2)):

                x = obj(req, 2)

                if req.isWriteReq:
                        #there are no definite reads, because the request might abort
                        for attr in defReadAttr(x,req) union mightReadAttr(x,req)    :
                                latestVersionBefore(x,attr,req.ts).pendingMightRead.add(req.id)
                else:

                        for w_req in writeWaitingQueue:
```

```
                    #if attr found which are going to be updated by this write req soon
                    if intersection of all attr in (mightReadAttr<x, req>) and (w_req.updates)
                    is non-empty:
                        # add this write_req id in the dependency list of read_req
                        if readWaitingQueue.contains(req):
                            readWaitingQueue.get(req).addInList(w_req)
                        else
                        # if list not present create the list and add write_req id
                            readWaitingQueue.put(req, List<wreq>)
                        #return since we can't move ahead for this read_req now,
                        #this req will be soon waked up by last exiting write_req
                    return

                # we know definite reads with their respective read timestamps
                for attr in defReadAttr(x,req):
                    latestVersionBefore(x,attr,req.ts).rts = req.ts
                # add might be read attributes to pending list with respective read timestamps
                for attr in mightReadAttr(x,req):
                    latestVersionBefore(x,attr,req.ts).pendingMightRead.add(req.id).



            req.worker = w #assign worker
            req.cachedUpdates[2] = cachedUpdates(x,req)
            send req to worker w

        On receiving <"restart", req> as coord(obj(req, 2)):

            #we know definitely it is a write request in case of restart tag

            for attr in defReadAttr(x,req) union mightReadAttr(x,req):
                v = latestVersionBefore(x,attr,req.ts)
                v.pendingMightRead.remove(req.id)


            x = obj(req, 2)
            req.ts = now()

            for attr in defReadAttr(x,req) union mightReadAttr(x,req)    :
                latestVersionBefore(x,attr,req.ts).pendingMightRead.add(req.id)

            req.worker = w #assign worker
            req.cachedUpdates[2] = cachedUpdates(x,req)
            send req to worker w
```

===

4. Worker

```
        On receiving req:

                #evaluatepolicy is the method which actually matches for rule in policy.xml and comes up
                #with decision with read and update attributes, the implementation for this black box for
                #this phase submission
                policyResult = evaluatePolicy(req)
                req.decision = policyResult.decision
                req.updates = empty

                if req.isWriteReq:
                        # index of obj, any one out of 1 or 2 otherwise -1
                        req.updatedObj = policyResult.updatedObj
                        # if updatedObj > 0, this is the index (1 or 2) of the other otherwise -1
                        req.rdonlydObj = policyResult.rdonlydObj
                        # set of updates on the updated obj as <attribute, value> pairs
                        req.updates = policyResult.updates

                if req.updatedObj == -1:
                        # req is read-only.
                        send <req.id, req.decision> to req.client
                        for i = 1..2:
                        send <"readAttr", req, i> to coord(obj(req,i))
                else:
                        # req updated an object.
                        send <"result", req> to coord(obj(req, req.updatedObj))

=======================================================
5.1 Coordinator
                On receiving <"readAttr", req, i>:

                        x = obj(req,i)
                        for attr in mightReadAttr(x,req):
                          v = latestVersionBefore(x,attr,req.ts)
                          v.pendingMightRead.remove(req.id)
                          # update timestamps for attr which have been read
                          if attr in req.readAttr[i]:
                                v.rts = req.ts

                        #traverse the coordinator level writeWaitingQueue and remove entry of exiting
                        #read_req also wake up this write req if current read_req is the only req in
                        #dependency list
                        for all entry e <w_req, List<e_reqId....> in writeWaitingQueue:
                                if e.list.contains(req) && e.list.size == 1:
                                        writeWaitingQueue.remove(e.req)
                                        #wake up write req buy rending it to current coordinator
                                        resend <"result", e.req> to coord(obj(e.req, e.req.updatedObj))
                                else if(e.list.contains(req)):
```

```
                        #else just remove the entry from list
                        e.list.remove(req)
```

==

5.2 Coordinator

```
        On receiving <"result", req> as coord(obj(req,req.updatedObj)):

            x = obj(req, req.updatedObj)

            #add to write waiting so that incoming read can check for conflicting attr
            writeWaitingQueue.add(req, List<empty>)

            conflict = checkForConflicts(req)
            if not conflict:

                #wait for requests which are still in readQueue, add them to dependency
                #queue
                for all <attr,val> in req.updates:
                    if latestVersionBefore(x,attr,req.ts).pendingMightRead is not
                      empty or contains entry other than req:

                        #add ongoing conflicting read only req to dependency
                        #list of this write req
                        writeWaitingQueue.get(req).addInList(r_reqId)

                        # return since you cant move ahead, this w_req will be
                        #waked up by last exiting conflicting r_req
                        return

                conflict = checkForConflicts(req)
                if not conflict:
                    send updates to the attribute database with timestamp req.ts
                    add updates to cachedUpdates
                    update data structure used by latestVersionBefore

                for attr in defReadAttr(x,req) union mightReadAttr(x,req):
                    v = latestVersionBefore(x,attr,req.ts)
                    v.pendingMightRead.remove(req.id)
                    if attr in req.readAttr[req.updatedObj]:
                        v.rts = req.ts

                  send <req.id, req.decision> to req.client_id
                  # notify coordinator of read-only object that req committed, so it can
                  # update read timestamps.
                  send <"readAttr", req, req.rdonlyObj> to coord(obj(req, req.rdonlyObj))
                else:
                  restart(req)
            else:
                restart(req)

            #remove self from coordinator level writeWaitingQueue and also remove entry
            #from waiting incoming read req present in readWaitingQueue
            writeWaitingQueue.remove(req)
            for all entry e <r_req, List<w_reqId....> in readWaitingQueue:
                if e.list.contains(req) && e.list.size == 1:
                    readWaitingQueue.remove(e.req)
                    #resend this read only request to current coordinator
                    resend e.req to coord(obj(e.req, current_coord))
                else if(e.list.contains(req)):
                    #else just remove id from dependecy list
                    e.list.remove(req)
```

==
6 Policy Independent Functions with implementation

```
    def checkForConflicts():
        for <attr, val> in req.updates:
        # note: if x.attr has not been read or written in this session, then
        # v is the special version with v.rts=0 and v.wts=0.
            v = latestVersionBefore(x,attr,req.ts)
            if v.rts > req.ts:
                return true
        return false

    def restart(req):
        #remove self from coordinator level writeWaitingQueue and also remove entry from
        #waiting incoming read req present in readWaitingQueue
        writeWaitingQueue.remove(req)
        for all entry e <r_req, List<w_reqId....> in readWaitingQueue:
            if e.list.contains(req) && e.list.size == 1:
                readWaitingQueue.remove(e.req)
                #resend this read only request to current coordinator
                resend e.req to coord(obj(e.req, current_coord))
            else if(e.list.contains(req)):
                #else just remove id from dependecy list
                e.list.remove(req)

        #send restart request to coordinator responsible for read only object
        send <"restart", req> to coord(req.rdonlyObj)

  #set of cached updates of attributes of x that are in defReadAttr(x,req) union mightReadAttr(x,req).
    def cachedUpdates(x,req):
        returnCacheAttr = {}
        objCache = cache.get(x)
        for attr in defReadAttr(x,req) union mightReadAttr(x,req):
            if objCache.has(attr):
                for <value, ts> in objCache.get(attr):
                    if ts < req.ts:
                        returnCacheAttr.add(<value, ts>)
                        break;

        return returnCacheAttr


    #latestVersionBefore(x,attr,ts) returns the most recent version of x.attr written before ts in
    #this session (i.e., since the coordinator process started). If no such version exists, then this
    #function returns a special version v with v.wts=0 and v.rts=0, representing the last version
    #written in the previous session; this special version is created on demand, when it is first
    #needed. Uses versionTable maintained at each coordinator level
    def latestVersionBefore(x, attr, ts):

        versionList = versionTable.get((x, attr))
        for i in range (0, versionList.size):
            if versionList.get(i).v.wts > ts:
                return versionList.get(i-1)
        #special version v with v.wts=0 and v.rts=0, representing the last version written in
        #previous session
        return new version(0, 0, [])

    #returns the object (subject or resource) whose coordinator should process the request first
    #(if i=1) or second (if i=2) the order in which the coordinators should process the request.
    def obj(req,i):
        if |mightWriteObj(req) == 1| :
            if i == 1:
                #choose element other than the might write object in case of i == 1
                #i.e we process read only obj first
                return req.objects - choose(mightWriteObj(req))
            else:
                #choose object which needs to be written in case of i == 2 i.e we process
                #this might write object after the read object.
                return choose(mightWriteObj(req))
        else:
            # send any of the two in set {sub, res}
            return req.objects[rand(0..1)]
```