Async Phase 3

Tamilmani Manoharan
Venkatakrishnan Rajagopalan

Pseudocode

```
HashMap versionMap<(obj,attr),List<Version>>
HashMap requests<reqId, Request>  # Map that stores all the original requests for restarting
HashMap recentUpdates<reqId, List<attr>> #Map store the updates that are going to be committed
HashMap cachedUpdates<obj,{attr,value}>

class Object {
   int id     # SubjectId or ResourceId
   HashMap Attr<name,value>
}

class Request{
   string subId
   string resId
   string action
   string reqId
   int order
   ReqType type
   ArrayList cachedUpdates<{attr,value}>[2]  # Set of cached updates for subject and resource
   ArrayList readAttr<List<string>>[2]  # List of attribute names
   int updateIndex
   int rdOnlyIndex
   HashMap updates<attr,value>   # attr name is String, value can be String or Int
}

enum ReqType {
   READ_REQ,
   WRITE_REQ,
   RESULT_RDONLY,
   RESULT_UPDATE,
   RESULT,
   UPDATEDB
}

class Message {
   ReqType type
   Request req
}

class Version {
   int rts
   int wts
}

def cachedUpdates(obj,req):
   cache = {}
   readAttr = defReadAttr(obj,req) union mightReadAttr(obj,req)
   updates = cachedUpdates[obj]
   for item in updates:
      if item.attr in readAttr:
         cache.add(item.attr,item.value)
   return cache

def latestVersionBefore(obj,attr,req.ts):
   versionList = versionMap[(obj,attr)]

   #sort version list based on timestamp
   sortedVersionList = sort(versionList)

   prev = {}
   for v in sortedVersionList:
      if(v.ts > req.ts)
         break
      prev = v

   v = prev
   if(v is empty)
      v.rts=v.wts=0
   return v

def evaluateRequest(req):
   updateIndex = -1
   updates = {}

   # check for all rules in policy files
   # change updateIndex to 1 or 2 if any one is getting updated, add changes to 'updates'

   readAttributes[1] = all attributes of 1 read so far
   readAttributes[2] = all attributes of 2 read so far

   result.decision = outcome of the policy check  # True or False
   result.readAttr = readAttributes
   result.updateIndex = updateIndex
   result.updates = updates

def checkForConflicts(req):
   obj = findObject(req,req.updateIndex)

   for <attr, val> in req.updates:
      # note: if obj.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(obj,attr,req.ts)
      if v.rts > req.ts:
        return true
   return false

def restart(req):
   originalReq = requests[req.reqId]
   obj = findObject(originalReq,1)
   coordinatorId = findCoordinator(obj)
   req.order=1
   sendRequest(originalReq,originalReq.type,coordinatorId)

Client() {
   Message msg
   while(msg = recvMsg())
   {
      if(msg.type == READ_REQ or WRITE_REQ)
      {
         req = msg.req
         req.type = msg.type
         obj = findObject(req,1)
         coordinatorId = findCoordinator(obj)
         req.order=1
         sendRequest(req,msg.type,coordinatorId)
      }
      else if(msg.type == RESULT)
         sendResultToApp(result)
}


Coordinator() {
   Message msg
   while( msg = recvMsg())
   {
      if(msg.type == READ_REQ or WRITE_REQ)
      {
         req = msg.req
         requests[req.reqId] = req  # Store original request, for retrieving and restarting later
         order = req.order
         obj = findObject(req,order)

         if(order == 1) {

            # To prevent starvation of write requests
            readAttr = defReadAttr(obj,req) union mightReadAttr(obj,req)
            for item in recentUpdates:
               await(no attr in item.attr is present in readAttr)

            req.ts = now()
         }

         if(msg.type == READ_REQ)
         {
            for attr in defReadAttr(obj,req):
               latestVersionBefore(obj,attr,req.ts).rts = req.ts

            for attr in mightReadAttr(obj,req):
               latestVersionBefore(obj,attr,req.ts).pendingMightRead.add(req.id)
         }
         else if(msg.type==WRITE_REQ)
         {
            for attr in defReadAttr(obj,req) union mightReadAttr(obj,req)
               v = latestVersionBefore(obj,attr,req.ts)
               v.pendingMightRead.add(req.id)
         }

         req.cachedUpdates[order] = cachedUpdates(obj,req)

         if(order==1)
            obj = findObject(req,2)
            coordinatorId = findCoordinator(obj)
            req.order=2
            send(req,msg.type,coordinatorId)
         else
            workerId = getWorker(obj)
            send(req,msg.type,workerId)
      }
      else if(msg.type == RESULT_RDONLY)
      {
         req = msg.req
         obj = findObject(req,req.order)
         for attr in mightReadAttr(obj,req)
```

```
          v = latestVersionBefore(obj,attr,req.ts)
          v.pendingMightRead.remove(req.id)
          if attr in req.readAttr[i]:
             v.rts = req.ts
      }
      else if(msg.type == RESULT_UPDATE)
      {
         obj = findObject(req,req.updateIndex)

         conflict = checkForConflicts(req)

         # Store recent updates to check for starvation in incoming new read requests
         forall <attr,val> in req.updates:
            recentUpdates[req.id].append(attr)

         if not conflict:

           # wait for relevant pending reads to complete
           await (forall <attr,val> in req.updates:
                  latestVersionBefore(x,attr,req.ts).pendingMightRead is empty
                  or contains only an entry for req)
           # check again for conflicts
           conflict = checkForConflicts(req)
           if not conflict:
             # commit the updates
             send(req,UPDATEDB,dbID)

             #create new version and append to version map
             forall <attr,val> in req.updates:
               cachedUpdates[obj].append(<attr,val>)
               Version newVersion = Version()
               newVersion.rts = 0
               newVersion.wts = req.ts
               versionMap[(obj,attr)].append(newVersion)

             # update read timestamps

             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

             #clearing the recent updates after committing
             recentUpdates[req.id] = None

             send(req,RESULT,req.clientId)

             obj = findObject(req,req.rdOnlyIndex)
             coordinatorId = findCoordinator(obj)
             send(req, RESULT_RDONLY, coordinatorId)
           else:
             restart(req)
         else:
           restart(req)
      }
   }
}

Worker() {
   Message msg
   while(msg=recvMsg())
```

```
  {
    req = msg.req
    result = evaluateRequest(req)
    req.decision = result.decision

    for i = 1 to 2:
      req.readAttr[i] = result.readAttr[findObject(req,i)]

    #For Read only requests, result.updates will be empty and result.updateIndex will be -1
    req.updateIndex = result.updateIndex
    req.updates = result.updates

    if(req.updateIndex == -1)
      send(req,RESULT,req.clientId)

      for i = 1 to 2:
        obj = findObject(req,i)
        req.order = i
        coordinatorId = findCoordinator(obj)
        send(req,RESULT_RDONLY,coordinatorId)
    else
      obj = findObject(req,req.updateIndex)
      coordinatorId = findCoordinator(obj)
      req.rdOnlyIndex = 3 - updateIndex  # since only 1 or 2 is possible
      send(req,RESULT_UPDATE,coordinatorId)
  }
}
```