CSE535 PHASE 3: PSEUDO-CODE

Team: Atanu Ghosh - 110280569, Muhammad Ali Ejaz - 110559131

Client

```
def run_client()
      LIST_OF_COORDINATORS = <list of all coordinators>
      // Read config.ini and generate list of requests and store in a
      // queue
      requestQueue = getSequenceFromConfigFile(configFile)

      // coordR denote the coordinator for the other object accessed by
      // req
      // coordW denotes the coordinator for the object in
      // mightWriteObj(req)
      while(!requestQueue.empty() && !timeoutQueue.empty())
            // timeoutQueue is checked to avoid a race condition
            // if last task in requestQueue gets dropped and times out
            // the timeout process will put the task back in
            // requestQueue. We need to have this while loop running
            // when that happens
            if (requestQueue.empty())
                  continue
            request = requestQueue.pop()
            coordR = hashFunction(request.obj1, LIST_OF_COORDINATORS)
            mightWrite = staticAnalysis(request)
            if (mightWrite == 1 && mightWriteOnObjAttr(obj1)) {
                  coordR = hashFunction(request.obj2,
                                                LIST_OF_COORDINATORS)
            }
            request.readOnly = (mightWrite == 0)
            timeoutQueue.add(request, now())
            request.fromClient = True
            send(request, to= coordR)


def staticAnalysis(request)
      return |mightWriteObj(request)|
```

```
def mightWriteOnObjAttr(object)
      if (mightWriteAttr(object))
            return true
      return false



receive(response, from= worker)
      if response.result = True
            // Allow access
      else
            // Deny Access
      timeOutQueue.remove(response.request)

def timeoutMonitor()
      timeoutDuration = <Read from Config>
      while(!timeoutQueue.empty() && !requestQueue.empty())
            // The following condition is used for the edge case
            // when the last task in timeoutQueue needs to get scheduled
            // due to timeout. Timeout queue will be empty and
            // requestQueue will have one task
            if (timeOutQueue.empty())
                  continue
            if (peek(timeoutQueue).ts - now() >= timeoutDuration)
                  requestQueue.add(timeoutQueue.pop())

run()
      run_client()
      timeoutMonitor()
```

```
// Request structure
Request {

      sessionID
      fromClient
      readOnly
}
```

Coordinator

```
// Coordinator class is same for all objects
def defReadAttr(x, req)
      // Global function to be called in setup to do static analysis of
      // policy and return data
      return dataList

def mightReadAttr(x, req)
      // Global function to be called in setup to do static analysis of
      // policy and return data
      return dataList

setup()
      versionCache = {}
      coordSessionID = UUId() // Session id for co-ordinator restarts
      PCA = {} // To handle starvation of write requests
      w = workers()


receive("evaluate", request, x, from= client/coord)
      x = obj(req,1/2)
      request[i].sessionID = coordSessionID

      if req.fromClient = true:
            req.ts = datetime.now()
```

```
if req.readonly == true: // read only is passed by client
      await(checkPCA() == False)
      for attr in defReadAttr(x,req):
            if req.fromClient:
                  latestVersion(x,attr).rts = req.ts
            else:
                  latestVersionBefore(x,attr,req.ts).rts = req.ts

      for attr in mightReadAttr(x,req):
            if req.fromClient:
                  latestVersion(x,attr).pendingMightRead.
                  add(<req.id,req.ts>).
            else:
                  latestVersionBefore(x,attr,
            req.ts).pendingMightRead.add(<req.id,req.ts>).

else: // if request is write
      for a in defReadAttr(x,req) union mightReadAttr(x,req)
            if req.fromClient:
                  v = latestVersion(x,attr)
            else:
                  v = latestVersionBefore(x,attr,req.ts)
            v.pendingMightRead.add(req.id).

req.cachedUpdates[1/2] = cachedUpdates(x,req)


if req.fromClient = true:
      // Hash function to get next coord
      req.fromClient = false
      send req to coord(obj(req,2/1))
else:
      // Choose worker
      req.worker = w
      send<"get_work_done", request, to= w)
```

```
receive("sendattr", req, i, x, from= worker/coord)
      if req.sessionID[i] != coordSessionID:
            // The coordinator has restarted in between.
            // So client must handle this based on timeout
            return

      x = obj(req,i)
      for attr in mightReadAttr(x,req)
            v = latestVersionBefore(x,attr,req.ts)
            v.pendingMightRead.remove(<req.id,_>)
            if attr in req.readAttr[i]:
                  v.rts = req.ts
```

```
receive("result", req, i, req.updatedObj, from= worker)
      if req.sessionID[i] != coordsessionID:
            // The coordinator has restarted in between.
            // So client must handle this based on timeout
            return

      // req updates the object that this coordinator is responsible
      // for check for conflicts.
      x = obj(req,req.updatedObj)
      req.rdonlyObj = req-req.updatedObj // The other object which is
                                         // read-only

      // check whether there are already known conflicts
      conflict = checkForConflicts(req)

      if not conflict:
            // Add to PCA to prevent starvation
            for attr in req.updates:
                  PCA[req[i].id][attr].append(req.id)
            // 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
              // send updates to the attribute database with
              // timestamp req.ts
              // send (req.updates, to= database)
              // add updates to cachedUpdates
             // update data structure used by latestVersionBefore

              // 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

              // Success in writing. Remove from PCA
              for attr in req.updates:
                    PCA[req[i].id][attr].remove(req.id)

              send <req.id, req.decision> to req.client
              // 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:
              // Write restarted. Remove from PCA
              for attr in req.updates:
                    PCA[req[i].id][attr].remove(req.id)
              restart(req)
  else:
        restart(req)
```

```
def checkForConflicts(req):
      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):
      req.fromClient = True
      // Hash function to get coord
      send <"evaluate", req, req.rdonlyObj> to coord(obj(req,
                                                      req.rdonlyObj))


def cachedUpdates(x, req):
      // Get the latest version before the current timestamp for all
      // attributes and store it in a list and send it back
      res = []
      for attr in (defReadAttr(x, req) union MightReadAttr(x, req)):
            res.append(latestVersionBefore(x, attr, req.ts))
      return res



def checkPCA(x):
      // Check for potentially conflicting attributes
      // This function is used to prevent write starvation
      For attr in defReadAttr(x,req) union mightReadAttr(x,req):
            If !PCA[x.id][attr].empty()
                  return True
            return False
```

```
def latestVersionBefore(x, attr, ts)
      try:
            return versionCache[x.id][attr].lower_bound(ts)[value]
      except:
            // If not found in cache create a new item in cache with
            // NULL value
            return  new versionCache[x.id][attr] = NULL marker


def latestVersion(x, attr)
      return latestVersionBefore(x, attr, now())
```

Worker instance

```
// Worker does the actual policy evaluation
setup()
      policy = readPolicyFromPolicyFile()


receive("get_work_done", request, from= coordinator)
      // evaluate req using the latest versions before req.ts (for
      // example, attribute database queries should have query
      // timestamp req.ts).
      result = evaluateRequest(request)
      request.decision = result.decision
      request.updatedObj = result.updatedObj
      if (result.updatedObj == 1)
            request.rdonlydObj = 2
      else if (result.updatedObj == 2)
            request.rdonlydObj = 1
      else
            request.rdonlydObj = -1

      // set of attribute updates to the updated object, if any,
      // represented as <attribute, value> pairs.
      request.updates = result.updatedAttributes

      for i in [1..2]:
            request.readAttr[i] = result.readAttr[Obj_i]
```

```
      if request.updatedObj = -1
        // request is read-only.
        send(request.id, request.decision, to= request.client)
        for i in [1..2]:
          send("readAttr", request, i, to= coord(obj(request,i)))
      else:
        // request updated an object.
        send("result", request, to= coord(obj(request,
                                              request.updatedObj)))

// 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 is discussed
// below.
obj(request, i)
      if (hashFunction(request.obj1.id, LIST_OF_COORDINATORS) == i)
            return request.obj1
      else if (hashFunction(request.obj2.id, LIST_OF_COORDINATORS)
                                                                  == i)
            return request.obj2


evaluateRequest(request)
      // evaluate req using the latest versions before req.ts (for
      // example, attribute database queries should have query
      // timestamp req.ts).
      Obj1Attr = latestVersionBefore(Obj1, request.ts)
      Obj2Attr = latestVersionBefore(Obj2, request.ts)

      // evaluate rules (uses policy files and DB)
      evaluation = evaluateRule(request)

      result.decision = evaluation.decision
      if (evaluate.Obj1Updated)
            result.updatedObj = 1
            result.updatedAttributes = Obj1UpdatedAttributes
      else if (evaluate.Obj2Updated)
            result.updatedObj = 2
```

```
      result.updatedAttributes = Obj2UpdatedAttributes
else
      result.updatedObj = -1

result.readAttr[Obj_1] = evaluation.readAttr[Obj_1]
result.readAttr[Obj_2] = evaluation.readAttr[Obj_2]

return result
```