CSE 535: Project Phase 4

Shahzeb Patel
110369918

Aadarsh Jajodia
110347086

Data Structures

```
Request {
        uuid
        Object1
        Object2
        action
        objecttype1
        objecttype2
        cachedUpdates[2]
        owner
        ts
}

Response {
        decision
        updates
        readAttr[2]
        updatedObject
        readOnlyObject
}

Version {
        name
        val
        rts
        wts
        pendingMightRead
}
```

Process classes:

NOTE: Garbage collection on attribute database and on the cache values is out of the scope for phase 3.

```
    # defReadAttr, mightReadAttr, mightWriteAttr
    # are supposed to be static analysis functions that will
    # use the data calcuated during static analysis
    # by client. These are static global functions available to
    # coordinator as well.

  global def defReadAttr(x, req)
        # retrieve populated data
        return data

  global def mightReadAttr(x, req)
```

```
        # retrieve populated data
        return data
```

class Coordinator {

```
    # cache of list of versions for every
    # attribute for an object.
    # version type is essentially an object with the rts
    # wts, value and pendingMightRead

    cache = {}

    # private workers. We use round robin approach to select a worker
    workers = []

    # Dictionary to store PCA
    potentialConflictAttrList = {}
```

run() {

```
        # Calculate a unique session id when session starts
        # If a session is restarted. it's session id has been changed

        session_id = uuid()

        while (exit != True)
    }
```

def latestVersionBefore(x, attr, ts):

```
        # should be latest but with wts less than ts

        # cache maintains a list of versions in decreasing order of their wts
        # of the version

        for version in cache[x][attr]:
            if (version.wts < ts)
                return version

            # no entry in cache
            return marker dummy version
```

def cachedUpdates(obj, req):

```
        # create a list of all versions for all attributes of x
        # such versions are latest but with wts less than the req.ts

        cachedResult[]

        attrList = defReadAttr(x, req) union mightReadAttr(x, req)

        for (attr in attrList):
            cachedResult.add(latestVersionBefore(x, attr, rs.ts)
```

def updateLocalCache(updateData, ts):
```
     x = updateData.obj
     updateList = updateData.list

     # Append the new version of attr to the list
     for attr in updateList:
            v = createVersion(x, attr.name, attr.val, ts)
            cache[x][attr].append(v)
```

def checkPCA(x, readAttrList):
```
      for attr in readAttrList:
          # Some write req is pending which is about to
          # update this attribute

          if (potentialConflictAttrList[x][attr].empty() == False)
              return True

      return False
```

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):
```
      send <"restart", req> to coord(obj(req, req.rdonlyObj))
```

def receive<"client">(request, client) {
```
      request.uuid = get_global_uuid()
      owner = client

      entrypoint(request)
  }
```

def receive<"restart">(req) {
```
      entrypoint(req)
  }
```

def entrypoint(req) {
```
      if (req.read):
          await (checkPCA(defReadAttr(x, req) union mightReadAttr(x, req)) == False)
      x = obj(req,1)

      req.ts = now()

      req.coord_id[1] = sessionid

      if (req.read):
          for attr in defReadAttr(x,req):
              latestVersionBefore(x,attr, req.ts).rts = req.ts

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

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

      send <"coord1">,req to coord(obj(req,2))
  }
```

def receive<"coord1">(req) {

```
      x = obj(req,2)

      req.coord_id[2] = sessionid

      if (req.read):
          for attr in defReadAttr(x,req):
              latestVersionBefore(x,attr, rs.ts).rts = req.ts

          for attr in mightReadAttr(x,req):
              latestVersionBefore(x,attr, rs.ts).pendingMightRead.add(<req.id,req.ts>).

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

      #choose worker w to evaluate this request

      req.worker = w

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

      send req to w
  }
```

def receive<"readattr">(response, req, i) {

```
      if req.coord_id[i] != session_id
```

```
            # ignore the request. Client will issue resend
            # after 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
    }
```

def receive<"worker">(response, req):

```
        if req.coord_id[req.updatedObj] != session_id
            # ignore the request. Client will issue resend
            # after timeout
            return

        # req updates the object that this coordinator is responsible for.
        # check for conflicts.

        x = obj(req,req.updatedObj)

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

        if not conflict:
            # wait for relevant pending reads to complete

            for attr  in req.updates:
                potentialConflictAttrList[x][attr].add(req.id)

            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()

            if not conflict:
                # commit the updates
                 send<response.updates, req.ts> to database
                updateLocalCache(response.updates, req.ts)

                # 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

                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))

                for attr in req.updates:
                    potentialConflictAttrList[x][attr].remove(req.id)

            else:
                for attr in req.updates:
                    potentialConflictAttrList[x][attr].remove(req.id)

                restart(req)
        else:
            restart(req)
```

Class Client

```
def setup_client(coordinator list, client_index, config) {
       # Reading the client config for this particular client index
       Request_list = config.read(“sequence_requests_for_client”)
       Timeout_interval = config.read(“timeout_interval_for_requests”)
       # In case sequence of requests is a random value, generate the random sequence of # requests
       using the random seed in the config file.
       if(Request_list == “random”)
       {
              # Generate a random request list using the random seed and already existing
              # sequence list using the config max_requests_to_generate
              Random_requests = []
              For i in range(0, max_requests_to_generate):
                     Random_requests.append(Request_list.append(i))
              Request_list = Random_requests
       }

       Total_requests = [] # List which appends each request
       # This same list is used when the client’s run function is called. One by one each
       # Generate the uuid for each request and populate any artificial delays that may
       # exist in the config file for each request.
       For request in request_list:
              Request.uuid = generate()
              Request.art_delay = config(“art_delay”)
              Total_requests.append(Request)
}

run_client():
{
    # For every client we ensure that once a request in sent, it waits for its response
    # before another request is sent for the same client.
    # For multiple clients we ensure that the clients send requests in the order in which
    # names have been read from the config file.
    # Sleep function is added to ensure that clients with lower index send their requests
    # earlier than clients with higher index.
    sleep(client_index/100)
    sendtask();
    # The client yields till the total number of requests are served.
    # This ensures that on receive calls are called for every client till the condition becomes
    true.
    # A separate thread is created to check if any of the requests have timed out.
    thread.create(timeout_check())

    await(num_requests == current_request_index)
}

/*
This function is run in a separate thread to keep checking for any requests that have timed out and
in case they have then that request is again send to the client
*/
timeout_check()
{
        while(1)
        {
                Now = datetime()
                if(Timeout_queue.top().timestamp - Now > timeout_interval)
                {
                        Timeout_queue.pop()
                        // Add this task back to the total_requests queue and send this task to the
                        coordinator.
                        Sendtask()
                }
        }
}


/*
Static Analysis which reads the policy file and populates mightWriteObj variable for a particular
action and resource type
mightWriteObject - This is a map with the action and resource type as the key. The value of this
map is as follows
        0 - None of the objects are updated.
        1 - Object 1 is updated.
        2 - Object 2 is updated.
        3 - Either of the two objects are updated.
*/
```

```
static_analysis_to_populate_might_write_object()
{
       Populate mightWriteObject(action, resource type)
}

/*
The function takes each request from the total_requests list for each client and sends that request
to the subject coordinator for this particular subject id.
*/
```

sendtask()

```
{
   Request = total_requests[current_request_index]
   # Send this request to the subject coordinator

   Timeout_queue.add(request)
   If mightWriteObject == 0 || mightWriteObject == 3  # Only reads are being performed so does not
matter or both the objects are being written so does not matter. We can send to any coordinator.
       send(request, coordinator_1)
   If mightWriteObject == 1
       send(request, coordinator_1)
   If mightWriteObject == 2
       send(request, coordinator_2)
}


/* The function receives the response from the subject coordinator once the request has been
processed. */
```

receive_object_coordinator(response, from_ = p)

```
{
   # Once we get a response for a particular request, we send the next request for this
particular client.
   Timeout_queue.remove(response.request)
   if(num_requests > client_request_number)
       sendtask()
   # Here if the timeout queue is empty as well as the request queue is empty which means all the
requests have been sent and their responses have been received then we join the thread, we had
created for handling the timeout functionality at the client.
}
```

Class Worker

```
/* This function setups up the worker process. It requires the coordinators list to send the policy
evaluation result to the respective coordinator, the database emulator object to make calls to the
object for retrieving the (attributes, values) corresponding to the subject and the resource and
the config object. It reads the policy xml file and populates the policy map used to evaluate the
policy.
```

setup_worker(coordinators_list, database, config)

```
{
    Policy_xml = config(“policy_xml_file”)
    Policy_map = {}
    read_policy(Policy_xml)
}


/* This function creates a dictionary of the policies in the xml based on the actions present in
the policy file. So a given action has a list of all rule objects that should be applied for that
given action
*/
```

def read_policy(policy_xml)

```
{
    # Parsing the policy XML here.
    For rule in polixy_xml:
        subject_condition = policy_xml.read(“subject_condition”)
        resource_condition = policy_xml.read(“resource_condition”)
        action = policy_xml.read(“action”)
        subject_update = policy_xml.read(“subject_update”)
        resource_update = policy_xml.read(“resource_update”)
        Rule_object = Rule(subject_condition, resource_condition, action,
                          Subject_update, resource_update)
        If action in Policy_map:
                Policy_map[action].append(Rule_object)
        Else
                Policy_map[action] = [Rule_Object]
}
```

def run_worker()

```
{
    # We await in the worker, till we receive an EXIT message from the coordinator.
    # Since coordinators are the process which start the workers, the respective coordinator
    # will send an exit message to all the workers which are associated with this coordinator.
    await(received(“EXIT”))
}

/*
Receives a request for evaluation of a policy from the resource coordinator and send the request to
the database to retrieve list of attributes, values that need to be fetched from database for the
evaluation of this policy.
*/
```

def receive_from_object_coordinator(request, from_ = p)

```
{
     Attribs = Get all attributes using latestVersionBeofre(req.ts)
     Updates, decision, attributes_read, updatedObj = evalute_policy(Attribs)
     req.decision = decision
     req.updatedObj = updatedObj
     if(updatedObj == -1) req.rdonlydObj = -1
     Else req.rdonlydObj = (updatedObj == 1)?2:1
     req.updates = Updates
     for i in [1..2]:
       req.readAttr[i] = attributes_read[i]
```

```
    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))
}
```

def evaluate_policy(attributes)
{
# Evaluates the policy based on the attributes.
}

```
/*
This function 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. i is the index of the coordinator.
*/
```

```
def obj(req, i):  {
   If hash(req.Object1) == i)
      Return req.Object1
   Else Return req.Object2
}
```