Gargi Saha(110923743)

Sagar Shah(110958554)

Multiversion Concurrent Evaluation of History-Based access control policies using timestamp ordering

'''

Design Decision

In order to prevent deadlocks, we have used a timer that times out after a random interval of time if the condition for await (that is to wait for pendingMightRead list of all attributes that are to be updated by this request to empty out) is not true by then. If a request is restarted once, when it arrives at its write coordinator for the second time, we place it in the pendingMightWrite list of its mightWriteAttr arribute list as a restarted request has high probability of being a write request, and every incoming request that wants to read that attribute will have to wait. There will be no scenario as per our design wherein a read or write request will starve.

Every time a coordinator starts up, it marks the beginning of a session using a timestamp. If a write request arrives from the worker with a timestamp lower than the current session's timestamp, then we do nothing and allow the client to time out.
'''

'''

Given functions

The following functions are populated based on static analysis of policy
and information avaliable in locally
'''

```
mightwriteobj(req) #subset of {req.sub, req.res}
defReadAttr(x, req)
mightReadAttr(x, req)
mightWriteAttr(x, req)
CoordList = [] #List of all Coordinators
```

'''

Other helper functions used

'''

```
def obj(req, i):
        return req.objects[i]


def coord(obj):
        # Some hash function used that maps objects to coordinators
        return CoordList[hash(obj.id)]
```

```
Class Versions:
    def __init__(self,rts=0,wts=0,value=None)
        self.rts = rts
        self.wts = wts
        self.value = value
        self.pendingMightRead = []
        self.pendingMightWrite = []


def cachedUpdates(x, req):
    cu = {}
    l = {attr: attr in mightReadAttr(x, req),attr in defReadAttr(x, req)}
    for attrs in l:
    '''
    if attrs is an attribute of an object handled by this coordinator
    get value of latest version of that attribute and return
    '''
        attrVersion = latestVersionBefore(x, attrs, req.ts)
        cu[attrs] = copy.deepcopy(attrVersion)
    return cu
```

```
'''
CACHE structure:

        CACHE:      {
                        object1id:      {
                                            attr1: [Version_1, Version_2, ...],
                                            attr2: [Version_1, Version_2, ...]
                                        },
                        object2id:      {
                                            attr1: [Version_1, Version_2, ...],
                                            attr2: [Version_1, Version_2, ...]
                                        }
                        .
                        .
                        .
                    }
'''
```

```
def latestVersionBefore(x, attr, ts):
    prevI = None
    if x in CACHE and attr in CACHE[x]:
        for i in CACHE[x][attr]:
            if(i.wts > ts):
                break
            prevI = i
    if prevI == None:
        '''
        prevI is None, means ts is old & required version is not in CACHE.
        We will add a special version that has rts=0 and wts=0
        and insert that to the start of list of versions for that attribute.
        This same special version is returned.
        '''
        if x not in CACHE:
            CACHE[x] = {}
        if attr not in CACHE[x]:
            CACHE[x][attr] = []
        CACHE[x][attr].insert(0, Versions())
        prevI = CACHE[x][attr][0]
    return prevI


def latestVersion(x, attr):
    if x in CACHE and attr in CACHE[x]:
        return CACHE[x][attr][len(CACHE[x][attr])-1]
```

Main Pseudo Code

```
class Request:
    '''
    the following dictionaries will contain all attributes of the resource
    and subject that the client/coordinator has any information about
    '''
    self.res = {}
    self.subj = {}
    self.action = None
    self.mightWriteObj = []
    self.objects = []

    def  __init__(subject,resource,action_type):
        self.subj = subject
        self.res = resource
        self.action = action_type
```

```
process Client:
    '''
    Client will send objects of type subject and resource based on information available to itself
    or the application, or a static analysis of policy.
    '''
    Request req = new Request(subject,resource,action_type)
    req.client = self
    if req.res in mightWriteObj(req):
        obj1 = req.subj
        obj2 = req.res
    else
        obj1 = req.res
        obj2 = req.subj
    req.objects.append(obj1)
    req.objects.append(obj2)
    send('Request',req,n=1,to=coord(obj(req,1)))

    def receive(msg=('Response', req)):
        # Application specific code
        print("Response", req)
```

```
process Coordinator:
    def __init__(self):
        self.sessionID = now()


    def receive(msg=('Request',req,n)):
        Object x = obj(req,n)
        if n == 1:
            req.ts = now()

        # Wait to avoid starvation
        toBeRead = defReadAttr(x, req) + mightReadAttr(x, req)
                  + mightWriteAttr(x, req)
        '''
        (*) If pendingMightWrite list of any attribute is nonempty, then this
        request must wait in order to prevent starvation of the restarted request.
        '''
        await (forall attr in toBeRead:
             latestVersionBefore(x,attr,req.ts).pendingMightWrite.empty())

        if (len(mightWriteObj(x, req)) == 0):
            # Readonly request
            for attr in defReadAttr(x, req):
                latestVersionBefore(x, attr, req.ts).rts = req.ts
        else:
            for attr in defReadAttr(x,req):
                latestVersionBefore(x, attr, req.ts).pendingMightRead.append(req.id)
            if req.restarted:
                '''
                If already restarted, then we can say with high probability that
                its a write request and add it to pendingMightWrite list of all those
                attributes in its MightWriteAttr list. Refer to (*)
                '''
                for attr in mightWriteAttr(x, req):
                    latestVersionBefore(x, attr, req.ts).pendingMightWrite.append(req.id)

        for attr in mightReadAttr(x, req):
            latestVersionBefore(x, attr, req.ts).pendingMightRead.append(req.id)

        req.cachedUpdates[n] = cachedUpdates(x, req)
        if n == 1:
            send('Request',req,n=2,to=coord(obj(req,2)))
        else:
            send('Request',req,to=Worker[hash(req.id)])

    def receive(msg=('ReadAttr', req, i)):
        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
        for attr in mightWriteAttr(x, req):
            v = latestVersionBefore(x, attr, req.ts)
            v.pendingMightWrite.remove(req.id)
```

```
def checkForConflicts(x, req):
    for <attr, val> in req.updates:
        v = latestVersionBefore(x,attr,req.ts)
        if v.rts > req.ts:
            return true
    return false


def receive(msg=('Result',req)):
    # Do nothing if the write request started in older session
    if req.ts < self.sessionID:
        return
    Object x = obj(req,req.updatedObj)
    # check whether there are already known conflicts
    conflict = checkForConflicts(x, req)

    if conflict == False:
        # wait for relevant pending reads to complete
        for <attr,val> in req.updates:
            latestVersionBefore(x, attr, req.ts).pendingMightWrite.append(req.id)
        '''
        An approach to prevent deadlocks:
        await condition is true if timeout timer expires or the pendingMightRead list of
        attributes to be updated, empties out.
        '''
        start_timer = now()
        timeout = random.random()
```

```
await(forall <attr,val> in req.updates :
        (latestVersionBefore(x,attr,req.ts).pendingMightRead.empty() == true) or
        (len(latestVersionBefore(x,attr,req.ts).pendingMightRead) == 1 and
              req.id in latestVersionBefore(x,attr,req.ts).pendingMightRead)), timeout = timeout)
if  now() - start_timer >= timeout:
       send(('Restart', req, 1), to=coord(obj(req,1)))
       send(('Restart', req, 2), to=coord(obj(req,2)))
       return

conflict = checkForConflicts(x, req)

if not conflict:
       send("UpdateDatabase",req.updatedObj,req.updates,req.ts,to=Database)
       for <attr,val> in req.updates:
              v = Versions(req.ts,req.ts,val)
              CACHE[x][attr].append(v)

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

       send(('Response', req), to=req.client)
       send(('ReadAttr', req, 1), coord(obj(req,1)))
```

```
            send(('ReadAttr', req, 2), coord(obj(req,2)))
        else:
            send(('Restart', req, 1), to=coord(obj(req,1)))
            send(('Restart', req, 2), to=coord(obj(req,2)))
    else:
        send(('Restart', req, 1), to=coord(obj(req,1)))
        send(('Restart', req, 2), to=coord(obj(req,2)))


def receive(msg=('Restart', req, n)):
    x = obj(req, n)

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

    for attr in mightWriteAttr(x, req):
        latestVersionBefore(x, attr, req.ts).pendingMightWrite.remove(req.id)
    if n == 2:
        req.restarted = True
        send(('Request', req, 1), to=coord(obj(req, 1)))
```

```
process Worker:
    def receive(msg = (req,) from_=p):

        # Do Static Analysis of policy and give all the rules that correspond
        # to {req.sub.type, req.res.type, action}
        rules = staticAnalysis(req)
        # all attributes of object 1 that will or might be accessed
        attrset1 = defReadAttr(obj(req, 1), req) + mightReadAttr(obj(req, 1), req)
        for rule in rules:
            for attrNeeded in getAttrNeeded(rule):
                if attrNeeded in attrset1:
                    n = 1
                else:
                    n = 2
                if attrNeeded in req.cachedUpdates[n] and
                  cachedUpdates[n][attrNeeded].value != None:

                    req.readAttr[n].append(cachedUpdates[n][attrNeeded])
                else:
                    v = readFromDatabaseVersionBefore(attrNeeded, req.ts)
                    req.readAttr[n].append(v)
            result = evaluate(rule, req)
            '''
            class Result
                self.decision    # deny or permit
                self.updatedObj # object that will be updated
                self.updates     # [<attr1, value1>, <attr2, value2>, ..]
            type(result) = Result
        '''
        if result.decision == 'permit' or result.decision == 'deny':
            break
if result.decision == None:
    req.decision = 'deny'
else:
    req.decision = result.decision

if result.updatedObj == None:
    # Read Request
    req.updatedObj = -1
    req.rdOnlyObj = -1
    req.updates = []
    send(('Response', req), to=req.client)
    for i in [1, 2]:
        send(('ReadAttr', req, n=i), to=coord(obj(req,i)))
else:
    # Write Request
    if result.updatedObj == obj(req, 1):
        req.updatedObj = 1
        req.rdOnlyObj = 2
    else:
        req.updatedObj = 2
        req.rdOnlyObj = 1
    req.updates = result.updates
    send(('Result', req), to=coord(result.updatedObj))
```