

1.Data Structure

For the MVCC implementation in the Phase 3, we add two more message classes EvalReqMvcc and EvalRespMvcc.

```
EvalReqMvcc:
# Basic fields same as Phase 1 and 2
  seq:      Sequence number indicating different request in test case
  id:       Global unique ID generated by coordinator
  app:      Request source (client)
  subject:  Subject ID
  resource: Resource ID
  action:   Action name
# Specific fields of MVCC
  ts:       Timestamp of current request
  cachedUpdates: piggybacked cached data in coordinator
```

```
EvalRespMvcc:
# Fields populated from Request
  seq, id, app, subject, resource, action, ts
# Specific fields of MVCC
  decision: Evaluation result
  readAttr: Attributes read
  updates:  Attributes updated
  updatedObj: Object updated
  rdonlyObj: Object read
```

2.Master

The only change of Master is the initialization of static analyzer and running mode (normal or mvcc) to distinguish code flow of Phase 3 from Phase 2. Pass analyzer and mode to client, coordinator and worker as needed.

Note: To make it clear, we keep the function name as provided and will make it consistent with existing functions later in term of naming convention.

3.Client (Application)

Client determine the objects possibly involved in write by the information available in current request and by using static analyzer (described in the following Coordinator section).

```
router:      Router that has knowledge of which coordinator takes care of the object
seqfrom:     First sequence number for which current client is responsible
seqto:       Last sequence number for which current client is responsible
cfg:         Configuration for client side
subjects:    Subject ID list (used for random payload generation)
resources:   Resource ID list
actions:     Action name list
analyzer:    Static analyzer that decides which object should process first
mode:        Running mode (oplock for Phase 1/2 or mvcc for Phase 3)
sentreq:     List of request that has been sent used for timeout check

def run():
  for i from seqfrom to seqto:
    # 0.Create request according to the running mode
    req = EvalReq() if mode == oplock else EvalReqMvcc()
    req.seq = i
```

```

# 1.Generate specific or random payload (req.subject, resource, action)
#   Since it's exactly the same as that in Phase 2, so we omit the detail here...

# 2.Send to correspondent coordinator
if mode == oplock:
    coord = router.get_sc(req.subject)
else:
    x = analyzer.obj(req, 1)
    coord = router.route(x)
send(('reqapp', req,), to=coord)
sentreq[i] = (req, coord, current timestamp)

# 3.Wait for exit command
await(received('done'))

def receive(msg=('respapp', seq, result,), from_=sc):
    Output decision or perform other operation
    Delete entry sentreq[seq]

# Re-submit request if timeout which is configured in config file
for seq, req, coord, ts in sentreq.items():
    if current timestamp - ts > cfg['timeout']:
        send(('reqapp', req,), to=coord)
        sentreq[seq] = (req, coord, current timestamp)

```

Analyzer (Static analysis helper class)

Static analyzer that preread policy rule to make decision of:

- Which object could possibly be involved in write
- Which object should be accessed first to reduce latency
- Is the request readonly?

```

policy: Analysis result of policy file
<(subject,resource,action),
  <rule-id, [subject-read-attrs,
            subject-update-attrs,
            resource-read-attrs,
            resource-update-attrs]>>

def parse_policy(policyfile, policy):
    tree = ET.parse(policyfile)

    # Analyze rules one by one
    for rule in tree.getroot().iter('rule'):
        subject = find subject ID attribute value
        resource = find resource ID attribute value
        action = find action name value

        # Analyze and save read/update attributes in each rule
        # cur is analysis result of current rule
        for attr, val in subject read attributes:
            cur[SUB_READ].append((attr, val))
        for attr, val in resource read attributes:
            cur[RES_READ].append((attr, val))
        if rule has subject update attributes:
            for attr, _ in subject update attributes:
                cur[SUB_UPD].append(attr)
        if rule has resource update attributes:
            for attr, _ in resource update attributes:
                cur[RES_UPD].append(attr)
        policy[(subject, resource, action)].append(cur)

# If written object is decided, choose it as second to process
# Otherwise (readonly request or more than 1 written object which cannot be decided).
# Choose subject as the first object to handle randomly.
def obj(req, i):
    wrtobj = mightWriteObj(req)
    if len(wrtobj) == 1:
        rdoobj = req.subject if req.subject != wrtobj[0] else req.resource
        return rdoobj if i == 1 else wrtobj[0]

```

```

    else:
        return req.subject if i == 1 else req.resource

def readonly(req):
    return not mightWriteObj(req)

def mightWriteObj(req):
    wrtobj = list()
    if policy[(req.subject, req.resource, req.action)][SUB_UPD]:
        wrtobj.append(req.subject)
    if policy[(req.subject, req.resource, req.action)][RES_UPD]:
        wrtobj.append(req.resource)
    return wrtobj

```

Router (Object routing helper class)

Same as that in Phase 2 which makes use of hash function to assign role to coordinator except add a new function to determine responsible coordinator without knowing object type in advance.

```

scmap = <subjectID, coordinator>
rcmap = <resourceID, coordinator>

# assign(subjects, resources, co), get_sc(sub), get_rc(res) exactly the same as before,
#so omit details here

def route(req, x):
    if x in scmap:
        return scmap[x]
    else:
        return rcmap[x]

```

4.CoordinatorMvcc

Due to significant changes in Coordinator component, we add a new class rather than mess up the original Coordinator in Phase 2.

```

router: Router that has knowledge of which coordinator takes care of the object
admin: Administrator of local state (cache and version)
idgen: Generator of global unique ID which is exactly same as Phase 2.
pendingUpdates: Save pending write request info for conflict check
rdonlyPendingQ: List of waiting readonly request coming after the pending write request

# Run and up until receiving stop command 'done'
def run():
    while True:
        await(received(('done')))

# Receive route table and subject/resource attribute list from master process
def receive(msg=('prepare', router)):
    self.router = router

# Coord1: Received request from Application
def receive(msg=('reqapp', req,), from_=p):
    # Pending readonly request to prevent pending write request starvation
    if analyzer.readonly(req) and might_conflict(req):
        rdonlyPendingQ.append(req)
    else:
        handle_request(req, 1)
        nextco = router.route(req, analyzer.obj(req, 2))
        send(('reqeval', req), to=nextco)

# Coord2: Received request from Coordinator 1
def receive(msg=('reqeval', req,), from_=p):
    handle_request(req, 2)
    worker = next(iter(workers))
    send(('assigneval', req), to=worker)

```

```

# CoordW: Receive evaluation result from worker
def receive(msg=('respeval', resp), from_=p):
    x = analyzer.obj(resp, resp.updatedObj)
    if not conflict(resp):
        # Wait for all pending read complete
        pendingUpdates[x] = resp.updates
        await(each((attr, val) in resp.updates,
            has= (not admin.latestVersionBefore(x, attr, resp.ts).pendingMightRead) or
                (admin.latestVersionBefore(x, attr, resp.ts).pendingMightRead
                 has only 1 entry which is related to resp)))
        resume_pending_request(resp)

    if not conflict(resp):
        # Commit update to database
        send(('writeattr', resp.updates), to=db)

        # Commit update to cache and update version
        admin.commit_cache(resp.updatedObj, resp.updates, resp.ts)
        admin.update_version(resp.updatedObj, resp.updates, resp.ts)
        update_read_ts(resp, resp.updatedObj,
            analyzer.defReadAttr(x, resp) union analyzer.mightReadAttr(x, resp))

        # Notify application the evaluation result
        send(('respapp', resp.seq, resp.decision), to=resp.app)

        # Notify other coordinator to update rts for attributes read
        coordR = router.route(resp, resp.rdonlydObj)
        send(('readAttr', resp, resp.rdonlydObj), to=coordR)
    else:
        restart(resp)
else:
    restart(resp)

# CoordR: Get notified that which attr in mightReadAttr is actually read
def receive(msg=('readAttr', resp, i), from_=p):
    x = obj(resp, i)
    update_read_ts(resp, i, analyzer.mightReadAttr(x, resp))

def handle_request(req, i):
    # 1.Generate unique global ID
    if req.id is None:
        req.id = idgen.next()
        req.ts = admin.now()

    # 2.Setup administration
    x = analyzer.obj(req, i)
    defR = analyzer.defReadAttr(x, req)
    mgtR = analyzer.mightReadAttr(x, req)
    if analyzer.readonly(req):
        for attr in defR:
            admin.latestVersionBefore(x, attr, req.ts).rts = req.ts
        for attr in mgtR:
            admin.latestVersionBefore(x, attr, req.ts).pendingMightRead.add((req.id, req.ts))
    else:
        for attr in defR | mgtR:
            admin.latestVersionBefore(x, attr, req.ts).pendingMightRead.add((req.id, req.ts))

    # 3.Populate data piggybacked to request
    req.cachedUpdates[i] = admin.cachedUpdates(x, req)

def update_read_ts(resp, i, attrs):
    x = obj(resp, i)
    for attr in attrs:
        v = admin.latestVersionBefore(x, attr, resp.ts)
        v.pendingMightRead.remove(resp.id)
        if attr in resp.readAttr[i]:
            v.rts = resp.ts

def conflict(resp):
    for (attr, _) in resp.updates:
        v = latestVersionBefore(x, attr, resp.ts)
        if v.rts > resp.ts:
            return True
    return False

```

```

# Check if upcoming readonly request might conflict with pending write request if there is any
def might_conflict(req):
    x = pendingUpdates[0]
    updates = pendingUpdates[1]
    defR = analyzer.defReadAttr(x, req)
    mgtR = analyzer.mightReadAttr(x, req)
    return (updates.keys() intersect (defR union mgtR)) is not empty

# Handle all pending readonly request after the waiting write request complete
def resume_pending_request(resp):
    x = analyzer.obj(resp, resp.updatedObj)
    for req in list(ronlyPendingQ):
        if analyzer.obj(req, 1) != x:
            continue
        ronlyPendingQ.remove(req)
        handle_request(req, 1)
        nextco = router.route(req, analyzer.obj(req, 2))
        send(('reqeval', req), to=nextco)
    delete pendingUpdates[x]

def restart(resp):
    req = EvalReqMvcc(resp)
    prevco = router.route(req, obj(req, 1))
    send(('reqapp', req), to=prevco)

```

AdminMvcc (State management helper class)

Meanwhile, we add a new class AdminMvcc for MVCC state management.

```

subcache: Subject cache that contains <obj, <attr, (value, timestamp)>>
rescache: Resource cache that contains <obj, <attr, (value, timestamp)>>
versions: Version map <obj, <attr, [v1, v2...]>> wherein each Version contains:
    rts (read timestamp),
    wts (write timestamp)
    pendingMightRead (pending queue for uncertain read request)
window: Inconsistent time window of attribute database

# By now we only keep the most recent update in cache and improve if necessary
def commit_cache(x, updates, ts):
    cache = subcache if x in subcache else rescache
    for attr, newval in updates.items():
        cache[x][attr] = (newval, ts)

def update_version(x, updates, ts):
    for attr, _ in updates.items():
        versions[x][attr].append(Version(ts, ts))

def now():
    return current timestamp

def cachedUpates(x, req):
    cache = dict()
    if x in subcache:
        cache = subcache[x]
    elif x in rescache:
        cache = rescache[x]

    # Remove entry if live longer than inconsitent window of attribute DB
    for attr, _, ts in cache:
        if now() - ts >= window:
            delete cache[attr]
    return cache

# Return a special Version(rts=0, wts=0) if no version in current session
# Otherwise return latest version before specified ts
def latestVersionBefore(x, attr, ts):
    if x not in versions:
        versions[x] = dict()

    if attr not in versions[x]:
        versions[x][attr] = list()
        versions[x][attr].append(Version(0, 0))

```

```

for v in versions[x][attr]:
    if v.wts < ts and latest.wts < v.wts:
        latest = v
return latest

```

DynAnalyzer (Dynamic analysis helper class)

Extends Analyzer class to improve by incorporating more analyzing approaches. Since functions here are relevant to specific rule, we omit them here and only present other irrelevant functions.

```

# Determine definite read attributes as those appear in every matched rules
def defReadAttr(obj, req)
    return predict(obj, req, operator.eq)

# Others that are uncertain to be affected would be considered as might read attributes.
def mightReadAttr(obj, req)
    return predict(obj, req, operator.lt)

def predict(obj, req, compare):
    rules = policy[(req.subject, req.resource, req.action)]
    cnt = create a bag of <attr, counter>
    idx = SUB_READ if obj == req.subject else RES_READ
    for rule in rules:
        for attr, _ in rule[idx]:
            cnt[attr] += 1

    # Only keep what meets to compare operator
    # eq(=) means: attribute appear in each rule
    # lt(<) means: attribute appear in some of the rules
    defR = list()
    for attr, c in cnt.items():
        if compare(c, len(rules)):
            defR.append(attr)
    return defR

```

5.Worker

```

class Worker(process):
    def setup(db:set):
        self.policy = None
        self.result = None

    def run():
        # Get policy from DB.
        p = next(iter(db))
        send(('getpolicy'), to=p)
        while True:
            await(received(()))

    def receive(msg=('assigneval', req), from_=c):
        # In case not received policy from database yet.
        if self.policy == None:
            await(received(('records'))))

        # Evaluate the request.
        resp, rule = evaluate(req)
        # After evaluation, resp.{decision, updatedObj, ronlyObj, updates, readAttr[1..2]} are set.

        if resp.updateObj == -1:
            # req is read-only.
            # send <req.id, req.decision> to req.client
            send(('evalapp', req.seq, resp.decision), to=req.client)
            for i = 1..2:
                # send <"readAttr", req, i> to coord(obj(req, i))
                send(('evalresp'), to=coord(obj(req, i)))
        else:

```

```

        # req updated an object.
        # send <"result", req> to coord(obj(req, req.updatedObj))
        send(('evalresp', resp), to=coord(obj(req, req.updatedObj)))

def receive(msg=('policy', policy), from_=p):
    # Received policy data from database.
    self.policy = policy

def receive(msg=('query', result), from_=db):
    # Received latest version from database.
    self.result = result

def evaluate(req:EvalReqMvcc):
    resp = EvalRespMvcc(req) # construct response based on request
    # init default values
    resp.decision = Decision.deny
    resp.updateObj = -1
    resp.rdonlyObj = -1
    resp.updates = emptyset
    matched_rule = None

    # communicate with database to get latest version
    send(('query', req.ts), to=next(iter(db)))
    while True:
        await((received('queryresult'))))

    for rule in policy.iter('rule'): # rule is an ElementTree instance
        # check if conditions satisfy the current rule
        should_try_next_rule = False # flag used to continue outter for loop

        # subject
        should_try_next_rule = not do_attributes_match(rule.find('subjectCondition'),
            self.result[SUB], req)
        if should_try_next_rule == True:
            continue
        # else: Subject condition matched.

        # resource
        should_try_next_rule = not do_attributes_match(rule.find('resourceCondition'),
            self.result[RES], req)
        if should_try_next_rule == True:
            continue
        # else: Resource condition matched.

        # action
        action_rule = rule.find('action').attrib['name']
        if action_rule == req.action:
            # all subject, resource, and action matched
            # rule found
            resp.decision = Decision.permit
            matched_rule = rule

            # Is req readonly or update 1 object?
            is_read_only = mightWriteObj(req) == None
            if is_read_only:
                resp.updates = emptyset
                resp.updateObj = -1
            else: # update 1 object
                resp.updates, resp.updateObj = fulfill_obligation(matched_rule, resp)
                resp.rdonlyObj = 3 - resp.updateObj
            break

    return (resp, matched_rule)

def do_attributes_match(condition:Element, attributes_record:dict, req:EvalReqMvcc):
    for attribute_name in iter(condition.attrib):
        if attribute_name not in attributes_record.keys():
            # keys do not match
            return False

    # keys match, then check values
    # record attributes of obj(req, 1 or 2) read during evaluation
    sub_i = 1 if obj(req, 1) == subject else 2
    res_i = 3 - sub_i
    if type(condition) == subjectUpdate:

```

```

        # subject
        req.readAttr[sub_i].insert(attribute_name)
    else:
        # resource
        req.readAttr[res_i].insert(attribute_name)

    value_condition = condition.attrib[attribute_name]
    value_record = attributes_record.get(attribute_name)
    # need to deal with different value forms
    # case 1: $subject.ATTRIBUTE or $resource.ATTRIBUTE
    # keep replacing it with value it points until constant
    while isinstance(value_condition, str) and value_condition.startswith("$"):
        dot_index = value_condition.index(".")
        attribute_key = value_condition[dot_index + 1 :]
        sub_or_res = value_condition[1 : dot_index]
        if sub_or_res == 'subject':
            value = req.subrattr[attribute_key]
        elif sub_or_res == 'resource':
            value = req.resrattr[attribute_key]

        value_condition = value

    # case 2: <constant or >constant
    if isinstance(value_condition, str) and value_condition.startswith("<"):
        num_str = value_condition[1:]
        if not num_str.isnumeric():
            # the attribute value is not a numeric string
            return False

        # valid, then compare them
        num_condition = int(num_str)
        if value_record >= num_condition:
            return False

    elif isinstance(value_condition, str) and value_condition.startswith(">"):
        num_str = value_condition[1:]
        if not num_str.isnumeric():
            # the attribute value is not a numeric string
            return False

        # valid, then compare them
        num_condition = int(num_str)
        if value_record <= num_condition:
            return False

    # case 3: constant
    elif value_condition != value_record:
        return False

# all passed
return True

def fulfill_obligation(rule:Element, resp:EvalRespMvcc):
    # Check if any obligation exists in matched rule.
    # return response.updates
    # change based on latest version
    if rule.has('subjectUpdate'):
        attrs_to_update = self.result[SUB]
        obligation_element = rule.find('subjectUpdate')
        obj = SUB
    elif rule.has('resourceUpdate'):
        attrs_to_update = self.result[RES]
        obligation_element = rule.find('resourceUpdate')
        obj = RES
    else:
        # No obligation
        return None

    if obligation_element != None:
        attributes = obligation_element.attrib
        for attribute_name in attributes:
            value_to_update = attributes[attribute_name]
            # different update value forms
            # replace $subject.ATTRIBUTE with original value
            while value_to_update != None and

```



```

        isinstance(value_to_update, str) and
        value_to_update.startswith("$"):
    dot_index = value_to_update.index(".")
    attribute_key = value_to_update[dot_index + 1 :]
    sub_or_res = value_to_update[1 : dot_index]
    if sub_or_res == 'subject':
        value = resp.subwattr[attribute_key]
    elif sub_or_res == 'resource':
        value = resp.reswattr[attribute_key]

    value_to_update = value

# case 1: ++ or --
if value_to_update == '++' or value_to_update == '--':
    # case 1.1: no such key exists -> create a new k,v pair with value 0
    # e.g. viewCount
    if not attribute_name in attrs_to_update:
        attrs_to_update[attribute_name] = 0
    attrs_to_update[attribute_name] += 1
    continue

# case 2: constant -> write (or overwrite) the value
attrs_to_update[attribute_name] = value_to_update

return attrs_to_update, obj

```

6.Database

```

class DB(process):
    def setup(policy_filename:str, record_filename:str,
              minDBlatency:str, maxDBlatency:str, workers:set):
        tree = ET.parse(policy_filename)
        self.rules = tree.getroot()
        tree = ET.parse(record_filename)
        record = tree.getroot()

        self.actions = []
        # save possible actions
        for rule in self.rules.iter('rule'):
            action_element = rule.find('action')
            action = action_element.attrib.get('name')
            if action != None:
                self.actions.append(action)

        self.versions = {}

        minDBlatency = int(minDBlatency)
        maxDBlatency = int(maxDBlatency)

    def run():
        while True:
            await(received())

    def receive(msg=('getpolicy'), from_=p):
        send(('policy', self.rules), to=p)

    def receive(msg=('getsubs'), from_=p):
        send(('subs', self.subrecords), to=p)

    def receive(msg=('getresos'), from_=p):
        send(('resos', self.resrecords), to=p)

    def receive(msg=('getacts'), from_=p):
        send(('acts', self.actions), to=p)

    def receive(msg=('write'), T, Q, from_=p):
        # T: transaction
        # Q: data item
        if TS(T) = W-timestamp(version(Q)):
            self.version(Q) = Q

```

```
elif TS(T) > R-timestamp(version(Q)):
    v = new Version(Q)
    self.versions.insert(v)

def receive(msg=('query', req_ts), from_=worker):
    latest_version = new Version(ts=0)
    for v in versions:
        if req_ts > latest_version.ts:
            latest_version = v
    return latest_version
```