Data Structures

```
Request {
       uuid
       subject_id
       resource_id
       action
       sub_cached_updates
       res_cached_updates
       owner
       timestamp
}

Response {
       request
       result
       sub_cached_attr_used
       res_cached_attr_used
       sub_db_attr_used
       res_db_attr_used
       sub_to_update
       res_to_update
}


Rule {
       subject_condition
       resource_condition
       action
       subject_update
       resource_update
}


Database_Response {
       request
       sub_database_attributes
       res_database_attributes
}
```

Process classes:

```
// Coordinator can be a subject coordinator or resource coordinator
// Logics for both of them are embedded in different functions according to
// the aim. The flow is interleaved.

Coordinator {

    sub_main_cache
    sub_tent_cache

    res_main_cache
```

```
response_queue

request_sequence

workers  // Processes


run() {
    while (exit != True)
}




attributes_in_sync(map1, map2)
{
    for every key in map1:
    {
        timestamp1 = map1[key][1]
        if key in map2
        {
            timestamp2 = map2[key][2]
            if(val_in_map1 != val_in_map2) {
                return False
        }
        else
            return False
    }
    return True
}

check_conflict(latest_cache_map, sent_cached_map, database_read_map)
{
    if  sent_cached_map == Empty:
            return attributes_in_sync(database_read_map, latest_cache_map)
    else
            return attributes_in_sync(sent_cached_map, latest_cache_map)
            && attributes_in_sync(database_read_map, latest_cache_map)
}

/*
subject-coordinator flow to assign the id, append tentative updates to the
request and invoke the resource coordinator. This is non blocking, it
invokes the resource coordinator and exits
*/
receive_from_client(request, client) {
    // need the timestamp to synchronize the result received from worker
    // who evaluated the policy
    request.uuid = get_global_uuid()
    owner = client
```

```
    entrypoint(request)
}

// Subject coordinator
entrypoint(request) {
    //Append cached updates if any
    if request.subject_id in sub_main_cache {
        request.sub_cached_updates = sub_main_cache[request.subject_id]
    }

    if request.subject_id in sub_tent_cache {}
        request.sub_cached_updates.update(sub_main_cache[request.subject_id])
    }

    // Process request in order
    request_sequence.add(request)

    send(request, resource_coordinator)
}

// Resource coordinator
receive_request_from_sub_cooord(request) {

    //Append cached updates if any
    if request.resource_id in res_main_cache {
        request.res_cached_updates = res_main_cache[request.resource_id]
    }


    send(reqest, worker)
}


// Subject coordinator
receive_respones_from_worker(response) {

    if (response.request.subject_id in request_sequence) {

        current_request_sequence = request_sequence[response.request.subject_id]

        if (current_request_sequence[0].timestamp < response.request.timestamp)
            // Tentative dependency: enqueueing
            response_queue.put((response.request.timestamp, response))
         }
         else
            process_response(response)
    } else
        process_response(response)

}
```

```
// Subject coordinator
process_response(response) {

    retval = True

    # Tentative updates got reverted. Panic!!!!!
    if (response.sub_cached_attr_used.empty() == False)
        if (response.request.subject_id not in sub_main_cache)
            retval = False
    else {
        if response.request.subject_id in sub_main_cache{
            retval = check_conflict(sub_main_cache[response.request.subject_id],
                                    response.sub_tent_attr_used,
                                    response.sub_db_attr_used)
        }
     }

    if (retval == True) {
        # Execute tentative updates for subject attributes
        if (response.sub_to_update.empty() == False) {
            current_tent_map = {}

            if response.request.subject_id in sub_tent_cache:
                current_tent_map =  sub_tent_cache[response.request.subject_id]

            for k,v in response.sub_to_update.items()
                current_tent_map[k] = (v, time.time())

                sub_tent_cache[response.request.subject_id] = current_tent_map
        }

        # Check for resource conclicts
        send(response, resource_coordinator)

    else {
        # Restart
        request_sequence[response.request.subject_id].popleft()
        entrypoint(response.request)
        extract_next_response()
     }
}

extract_next_response() {

    if (response_queue.empty() == False){

        response = response_queue.get()[1]

        if (response.request.subject_id in request_sequence):
            current_request_sequence = request_sequence[response.request.subject_id]
```

```
            if (current_request_sequence[0].timestamp < response.request.timestamp):
                response_queue.put((response.request.timestamp, response))
            else:
                process_response(response)
        else:
            process_response(response)
    }

}
```

```
// Resource coordinator
receive_response__resource_coordiniator_conflict_check(response, subject_coordinator) {
    retval = True

    if response.request.resource_id in res_main_cache {
        retval = check_conflict(
                    res_main_cache[response.request.resource_id],
                    response.res_tent_attr_used,
                    response.res_db_attr_used)
    }

    if (retval == True) {

        if (response.res_to_update.empty() == False) {
            # Execute tentative updates for resource attributes
            current_tent_map = {}
            if response.request.resource_id in res_main_cache
                current_tent_map =  res_main_cache[response.request.resource_id]

            for k,v in response.res_to_update.items()
                current_tent_map[k] = (v, time.time())

            res_main_cache[response.request.resource_id] = current_tent_map

            # Commit resource updates to db as well
            commit_to_db = {}
            commit_to_db[str(response.request.resource_id)] = response.res_to_update
            send(commit_to_db, database)
        }
    }
    # Need to send result to subject in either conflict or no conflict case

    send(response, retval, subject_coordinator)
}

// Subject coordinator
```

```
    receive_response_subject_coordinator_conflict_result(response,
                            retval, subject_coordinator) {

        if (True == retval) {
            # No conflict, propagate the tentative subject updates to main

            # Propagate subject tentative updates to main cache

            if (bool(response.sub_to_update) == True) {
                current_sub_map = {}

                if response.request.subject_id in sub_main_cache:
                    current_sub_map = sub_main_cache[response.request.subject_id]

                current_sub_map.update(sub_tent_cache[response.request.subject_id])

                sub_main_cache[response.request.subject_id] = current_sub_map

                # Revert the sub_tent cache for this id
                sub_tent_cache[response.request.subject_id] = {}

                # Doing database updates
                send_sequence =  send_sequence + 1
                commit_to_db = {}
                commit_to_db[str(response.request.subject_id)] = response.sub_to_update
                send(('FROM_COORDINATOR_ATTR_UPDATE', commit_to_db), to = (database))
            }

            send_sequence =  send_sequence + 1
            send(response, response.request.owner)

            request_sequence[response.request.subject_id].popleft()
            # Process next response that was queued
            extract_next_response()
        } else {
            # Restart because of resource conflicts

            if response.request.subject_id in sub_tent_cache:
                sub_tent_cache[response.request.subject_id] = {}

            request_sequence[response.request.subject_id].popleft()

            entrypoint(response.request)

            extract_next_response()
        }


// Master
main() {
```

```
    # creating database process
    database = new(Database, num = 1)
    start(database)

    # creating coordinators
    coordinators_set = new(Coordinator, num = total_coords)

    for p in coordinators: setup(p, (coordinators, database, config, ))
    start(coordinators)

    # creating clients

    clients = new(Client, num = total_clients)
    i = 0
    for p in clients :
        setup(p, (coordinators, i + 1, config,))
        i = i + 1

    start(clients)

    for c in clients:  c.join()

    send(('EXIT'), coordinators)
    send(('EXIT'), database)
  }
```

Class Client

```
setup_client(coordinator list, client_index, config)
{
       // Reading the client config for this particular client index
       Request_list = config.read(“sequence_requests_for_client”)

       // 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.
   await(num_requests == current_request_index)
}

/*
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

   Coordinator_id = coordinators(rid % len(coordinators))
   send_subject_coordinator(request, coordinator[Coordinator_id])
}

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

receive_subject_coordinator(response, from_ = p)
{
   // Once we get a response for a particular request, we send the next request for this
particular client.
   if(num_requests > client_request_number)
       sendtask()
}
```

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
*/
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]
}

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.
*/
receive_from_resource_coordinator(request, from_ = p)
{
    send(request, to = database)
}

/* This method received the attributes required to evaluate the policy for this request.
```

```
*
receive_from_database(db_response)
{
   // If the request has an artificial delay specified, then we sleep for that time.
   // We create two maps.
   // Each key in this map is a tuple value, where the first element in the tuple is the value
      of the attribute and the second element is a flag which indicates this attribute has been
      Read from the database or from the tentative attributes list.
   sub_attribute_list_to_evaluate_policy = {}
   res_attribute_list_to_evaluate_policy = {}

   For key, value in db_response.request.sub_tent_updates:
      sub_attribute_list_to_evaluate_policy[key] = (value[0], 0)
   For key, value in db_response.request.res_tent_updates:
      res_attribute_list_to_evaluate_policy[key] = (value[0], 0)
   for key, value in db_response.sub_database_attributes
      sub_attribute_list_to_evaluate_policy[key] = (value, 1)
   for key, value in db_response.res_database_attributes
      res_attribute_list_to_evaluate_policy[key] = (value, 1)

   // Now we get the list of Rule Objects which satisfy this action.
   If action not in policy_map:
      Response.result = False // This policy evaluates to false by default since no entry for
                              // this action exists in the policy file.
      send_subject_coordinator(response, subject_coordinator)
   Else:
      Rules_list = policy_map[action]

   // Now for every rule, we call a validate method which first validates the attributes values
   // for the subject, and then validates the attribute values for the resource, if the values
   // If the values are in sync, then we call the update method to update the attributes as
   // specified by the update condition of the policy xml.

   Result, sub_attributes_from_cache, sub_attributes_from_db = validate_subject()
   if(Result == True)
   {
      Result, res_attributes_from_cache, res_attributes_from_db = validate_resource()
      if(Result == True)
      {

            // Populate the response object with the subject to update maps
            // and the resource to update maps.
            Response.sub_to_update = update_subject_attributes_after_policy_evaluation()
            Response.res_to_update = update_resource_attributes_after_policy_evaluation()
            Response.sub_tent_attr_used = sub_attributes_from_cache
            Response.sub_db_attr_used = sub_attributes_from_db
            Response.res_tent_attr_used = res_attributes_from_cache
            Response.res_db_attr_used = res_attributes_from_db
      }
   }
   send_subject_coordinator(Response, subject_coordinator())
```

```
}

validate_attributes_in_policy(rule_condition, attributes_list_to_evaluate_policy,
              request_tentative_updates, sub_attribute_list_to_evaluate_policy,
              res_attribute_list_to_evaluate_policy)
{
   // If no condition exists for this rule in the policy xml file return False
   If len(rule_condition) == 0 or len(attributes_list_to_evaluate_policy) == 0:
      Return (False, None, None)
   count = 0
   attributes_used_from_tent = {}
   attributes_used_from_db = {}
   For every key in rule_condition:
      If key in attributes_list_to_evaluate_policy:
             // Here we check if $ is present in the policy file.
             Rule_condition[key] = evaluate$values()
             if(attrbutes_values_match())
                    Count++;

   If count != number_of_attributes in rule_condition
      Return False, None, None
   Else:
      for every key in rule_condition
             // If this value is read from the cache.
             if attributes_list_to_evaluate_policy[key][1] == 0:
                    attributes_used_from_tent[key] = request_tentative_updates[key]
             Else:
             // Value is read from the database.
                    attributes_used_from_db[key] = db_attributes_map[key][0]

   Return (True, attributes_used_from_tent, attributes_used_from_db)
}

does_attribute_value_satisfy_condition(value_in_policy_file, value_from_db_or_cache)
{
   if( value_in_policy_file == empty and value_from_db_or_cache == empty)
      Return True // The values match
   if( value_in_policy_file == empty and value_from_db_or_cache != empty)
      Return False
   if( value_in_policy_file has < or > operators in it)
   {
      // Apply the respective operator < or > on the value_from_db_or_cache and check if it
      // Satisfies the condition. If it does then return True
   }
   if( value_in_policy_file != value_from_db_or_cache)
      Return False
   Return True
}

/*
This function updates the attribute value after a particular policy rule has been satisifed.
```

```
*/
update_attributes_after_policy_evaluation(update_condition, attribute_list_to_evaluate_policy,
             Sub_attribute_list_to_evaluate_policy, res_attribute_list_to_evaluate_policy)
{
   // This function updates the values as specified by the update condition in the policy xml.
   for every key, value in update_condition:
      If value == “++”
             To_update[key] = attribute_list_to_evaluate_policy[key] + 1
      Else If value == “--”
             To_update[key] = attribute_list_to_evaluate_policy[key] - 1
      Else If value starts with “$”
             To_update[key] = evaluate(update_condition[key,
                    Sub_attribute_list_to_evaluate_policy, res_attribute_list_to_evaluate_policy)
      Else
             To_update[key] = update_condition[key]
}
```

Database

```
setup_database(config)
{
   mindblatency = config("mindblatency")
   maxdblatency = config(“maxdblatency")
   randvalue = random.randrange(mindblatency, maxdblatency)
}

run_database()
{
   // Initializing the database object here by reading the values from the database file.
   Database = populate_db(policy_xml)
   await(received(“EXIT”))

   // Once we get a exit command to exit the database process, we dump all the database entries
   // to the log file.
   database.dump()
}



/* This function populates the database response, which contains the attributes which are present
in the database but not in the tentative attributes map received in the request
*/

receive_worker(request)
{
   // Create a database Response and populate 2 maps, subject_database_attributes and
   // resource_database_attributes. This essential does a set difference of db entries and cache
   // entries and returns keys which are present in db and not in cache.
    sub_attribute_diff = set(subject_database_attribute_keys
                           set(request.sub_tent_updates.keys()))
```

```
    for every attr in sub_attribute_diff
      db_response_sub_attribute_map[attr] = self.database[str(request.subject_id)][attr]

    res_attribute_diff = set(resource_database_attribute_keys
                          set(request.res_tent_updates.keys()))
    for every attr in res_attribute_diff
      db_response_res_attribute_map[attr] = self.database[str(request.resource_id)][attr]

    db_response.sub_database_attributes = db_response_sub_attribute_map
    db_response.res_database_attributes = db_response_res_attribute_map
    send_worker(db_response)
}

/*
This function receives from both the resource and the subject corordinator the attributes that need
to the flushed to the database and starts a thread which calls a function (after a certain random
time between mindblatency and maxdblatency) to commit the update values to the database
*/

receive_coordinator(attributes_to_update):
{
   threading.Timer(randvalue, commit_to_db, [attributes_to_update]).start()
}

commit_to_db(attributes_to_update)
{
   res_sub_id, value = attributes_to_update.popitem()
   self.database[res_sub_id].update(value)
}
```