CSE 535 Asynchronous Systems
Phase 2

Hari Prasath Raman, 110283168
Arvind Ram Anantharam, 110283876

October 16, 2016

1 Algorithm for Concurrency Control using Distributed Coordinator

Based on the publication:

Maarten Decat, Bert Lagaisse, Wouter Joosen. Scalable and Secure Concurrent Evaluation of History-based Access Control Policies. Proceedings of the 31st Annual Computer Security Applications Conference (ACSAC 2015). ACM, 2015.

1.1 Global Constants

```
# Message types

APP_EVALUATION_REQUEST
APP_EVALUATION_RESPONSE
SUB_EVALUATION_REQUEST
RES_EVALUATION_REQUEST
RES_COMMIT_REQUEST
RES_COMMIT_RESPONSE
WORKER_EVALUATION_RESPONSE
```

1.2 Application Instance

Psuedo Code

```
    # Map containing process information which is responsible for a
    # particular subject coordinator

    sub_coord = map()
```

```
# Method to evaluate the policy rules for a given subject and
# resource asynchronously
authorize(sub, res):
    sub_coord_id = sub_coord[id]
    timestamp = curr_time
    app_id = uuid()
    send_policy_eval_message(APP_EVALUATION_REQUEST, app_id,
                             sub, res, timestamp, sub_coord_id)

    # Patiently wait for the message with evaluation result
    result = wait_for_evaluation_result(APP_EVALUATION_RESPONSE)
    return result
```

1.3 Subject Coordinator

Psuedo Code

```
# Map containing process information which is responsible for
# a particular resource coordinator
res_coord = map()

# Maps sub id to attributes
#
# Why we need this ?
# To store the results of sub attr till data gets synced in
# distributed db
attr_cache = map()

# Maps eval id to app id and actual request tuple
app_req_map = map()

res_commit_q = map()
# Attribute cache expiry time in secs
#
# This value can be set based on the time taken for the
# distributed db to sync data
ATTR_CACHE_EXPIRY = 10

# Why Table ?
# Since, we need to lookup based on multiple parameters of the
# evaluation requests like eval_id, timestamp, sub_id storing
# them in table will make it easy to query
# Table schema for storing tentative data and clearing them
-------------------------------------------------------------------------
 | eval_id | timestamp | sub_id | sub | res_id | res | dependent_eval_ids |
-------------------------------------------------------------------------
-------------------------------------------------------------------------
 status | result |
```

```
-------------------------------------------------------------------------------
# eval_id - Primary Key
# status(values) - PENDING / WORKER_COMPLETE
eval_cache = table()

get_eval(eval_id):
    # Returns corresponding record tuple from eval_cache table
    eval = eval_cache[eval_id]
    return eval

get_app(eval_id):
    return (app_req_map[eval_id][app_id], app_req_map[eval_id][app_p])

setup_cache(eval_id, sub, res, action, dependent_eval_ids, timestamp status):
    # We will record the eval_id, updated subject attributes,
    # request's timestamp for the given subject in evaluation

    eval_cache.append(eval_id, timestamp, sub.id, sub.attrs,
                      res.id, res.attrs, dependent_eval_ids, status)

# Delete the record from eval_cache table
clear_cache(eval_id):
    eval_cache.delete(eval_id)
    del app_req_map[eval_id]

# Tentatively commit sub attr updates
update_cache(eval_id):
    eval_cache.update_status(eval_id, WORKER_COMPLETE)

# update dep evals when we detect some changes to dep evals
update_dep_evals(eval_id, dep_eval_ids):
    eval_cache.insert(eval_id, dep_eval_ids)

get_previous_committed_evals(timestamp):
    evals = eval_cache.query(row[status] == WORKER_COMPLETE and row[timestamp
    return evals

add_tentative_attr_updates_to_req(sub, timestamp):
    # find all evals which has status as WORKER_COMPLETE before
    # timestamp and update the corresponding subject attrs
    # for the current subject with those updates
    tentative_evals = get_previous_committed_evals(timestamp)

    # Update attrs with tentative values
    sub.attrs.update(tentative_evals.sub.attrs)
    for all attrs in received sub:
        key = (sub[type], sub[id], sub[attr])
        sub.attrs.update(attr_cache[key])
```

```
    return sub, tentative_evals.eval_ids

get_tentative_evals(attrs, timestamp):
    # Query eval_cache table for records whose timestamp is
    # lesser than input timestamp, has different values for
    # subject attributes used by record and has status as
    # WORKER_COMPLETE

    evals = eval_cache.query(row[status] == WORKER_COMPLETE and row[timestamp

    for each eval from evals:
        eval.add_corresponding_attrs_from_received_sub

    return evals_with_corresponding_sub_attrs

# returns the evals which are dependent on the given eval_id
get_evals_dependent_on(eval_id):
    eval = get_eval(eval_id)
    evals = eval_cache.query(row[req_no] > eval[req_no])

    for eval in evals:
        if eval_id in evals[dep_eval_ids]:
            result.append(eval)

    return result

# detects conflicts with tentative evals
has_subject_attr_updates(eval_id):
    eval = get_eval(eval_id)
    tentative_evals = get_tentative_evals(eval.sub.attrs,
                                          eval.timestamp)

    # Checks for attr value modification in the mean while
    return tentative_evals.sub.attrs != eval.attrs

restart(eval_id):
    eval = get_eval(eval_id)
    app_id, orig_req = app_req_map[eval_id]
    clear_cache(eval_id)
    send_policy_eval_message(APP_EVALUATION_REQUEST,
                             app_id, orig_req.sub,
                             orig_req.res, orig_req.timestamp, self)

evaluate(app_id, sub, res, timestamp):
    # Assign global unique id for this evaluation request
    eval_id = uuid()
    orig_req = (sub, res, timestamp)
    app_req_map[eval_id] = (app_id, orig_req)
    sub, dependent_eval_ids =
```

```
                    add_tentative_attr_updates_to_req(sub, timestamp)
    setup_cache(eval_id, sub, res, timestamp, dependent_eval_ids, PENDING)
    res_coord_id = res_coord[res.id]
    send(RES_EVALUATION_REQUEST, eval_id,
         sub, res, timestamp, res_coord_id)

process_worker_response(result, eval_id):
    if has_subject_attr_updates(eval_id):
        restart(eval_id)
    else:
        update_cache(eval_id)
        curr_eval = get_eval(eval_id)
        tentative_evals = get_tentative_evals(curr_eval.sub.attrs,
                            curr_eval.timestamp)
        # Ensures commit is issued in order of requests received
        for eval in tentative_evals:
            result = wait_for_completion(eval.eval_id)
            if result == FAILURE:
                # Restart self
                restart(eval_id)

        send(RES_COMMIT_REQUEST, eval_id, curr_eval.res,
            res_coord[curr_eval.res.id])

# For processing the acknowledgement from resource coordinator
process_resource_commit_response(eval_id, status):
    if status == SUCCESS:
        eval = get_eval(eval_id)

        # Ensures all the previous evals gets committed in order
        previous_evals = get_previous_evals(eval.timestamp)
        for eval in previous_evals:
            wait_for_completion(eval.eval_id)

        #updates the distributed attr db
        update_attr_db(eval.sub.attrs)

        # time-bound key
        attr_cache[sub.id] = eval.sub.attrs
        app_id, orig_req = app_req_map[eval_id]
        clear_cache(eval_id)   # Clear cache
        send_evaluation_result_to_app(APP_EVALUATION_RESPONSE,
                                      status, app_id)

    else if status == FAILURE:
        # Get all evaluations whose timestamp is greater
        # than eval_id's timestamp and eval_id in dependent_eval_ids
        evals = get_evals_dependent_on(eval_id, sub.attrs)
        # Restart all the dependent evaluations
```

```
            for eval in evals:
                restart(eval.eval_id)

            #restart myself
            restart(eval_id)

    # Main process which listens for the messages
    subject_coord():
        while(True):
            msg_type, data = receive()

            if msg_type == APP_EVALUATION_REQUEST:
                evaluate(data.app_id, data.sub, data.res, data.timestamp)

            else if msg_type == WORKER_EVALUATION_RESPONSE:
                process_worker_response(data.result, data.eval_id)

            else if msg_type == RES_COMMIT_RESPONSE:
                process_resource_commit_response(data.eval_id, data.status)
```

1.4 Resource Coordinator

Psuedo Code

```
res_cache = map()

# Maps res id to attributes
#
# Why we need this ?
# To store the results of res attr till data gets synced in
# distributed db
attr_cache = map()

# Attribute cache expiry time in secs
#
# This value can be set based on the time taken for the
# distributed db to sync data
ATTR_CACHE_EXPIRY = 10

setup_res_attr(res):
    # Update res[attrs] with recent attributes stored in cache
    key = (res[type], res[id])
    res[attrs] = res_cache[key]
    return res

update_cache(res):
```

```
        key = (res[type], res[id])
        res_cache[key] = res[attr]

    get_eval(eval_id):
        # Returns corresponding record tuple from eval_cache table
        return eval

    # Checks for conflicts by comparing old attribute values with current
    # evaluation's attribute values
    conflict_exists(res):
        key = (res[type], res[id])
        if key not in res_cache:
            return False
        return res_cache[key][attrs] != res[old_attr][attrs]

    assign_worker(eval_id, sub, res):
        # Assign worker in a round robin fashion
        nex_worker_idx += 1
        worker = workers[nex_worker_idx \% len(workers) ]
        send(WORKER_EVALUATION_REQUEST, eval_id, sub, res, worker_id)

    evaluate(eval_id, sub, res, timestamp):
        if eval_id exists in eval_cache:
            # clear administration for current evaluation
            clear_cache(eval_id)

        # setup administration
        setup_cache(eval_id, sub, res, timestamp)
        assign_worker(eval_id, sub, res)

    commit_eval(app_id, eval_id, res):
        # res here is the current res data received from subject
        # coordinator
        eval = get_eval(eval_id)
        if conflict_exists(res):
            send(RES_COMMIT_RESPONSE, FAILURE, eval_id, eval.sub.coord_id)
        else:
            #updates the distributed attr db
            update_attr_db(res_attrs)
            update_cache(res)
            send(RES_COMMIT_RESPONSE, SUCCESS, eval_id, eval.sub.coord_id)

    resource_coord():
        while(True):
            msg_type, data = receive()
            if msg_type == SUB_EVALUATION_REQUEST:
                evaluate(data.eval_id,
                         data.sub,
                         data.res,
```

```
                        data.timestamp)

            else if msg_type == RES_COMMIT_REQUEST:
                commit_eval(data.eval_id, data.res.attrs)
```

1.5 Worker

Psuedo Code

```
evaluate():
    # Execute all the valid policy evaluations for corresponding
    # subject and resource
    # read subject and resource attr values from db
    sub, res = read_from_db(sub_attrs, res_attrs)
    result = execute_policy(sub, res)
    return result

evaluate_policy():
    # Keep listening for policy queries
    while(True):
        # Blocking call for receiving policy query
        # Receives only the evaluation request from
        # resource coordinator
        eval_id, sub, res = receive(WORKER_EVALUATION_REQUEST)
        result = evaluate(sub, res)
        send(WORKER_EVALUATION_RESPONSE, result, eval_id, sub.coord_id)
```