Distributed Simulation

Optimistic Algorithm

*Thomas Laverghetta (*[*tlave002@odu.edu*](mailto:tlave002@odu.edu)*)*,   
Old Dominion University – Computational Modeling and Simulation Engineering

Please read over HW 5 System Requirements document (which accompanies this report) first to gain understanding of program overview.

# Algorithms

An optimistic algorithm is a distributed simulation executive algorithm which describes the behavior of continuously processing events (i.e., not waiting for all processes to send messages (msgs) before processing). Therefore, if a msg is received (recv’d) with time less than current simulation time, then rollback will have to occur. Rollback is a correction mechanism for reversing time. During rollback, executed events will be reversed until event with timestamp less than recv’d time is found. During the reversal of events, to account for events scheduling new events, anti-msgs will be created and sent. Anti-msgs eliminate scheduled future events and rollback executed events.

To create the optimistic algorithm, the following must be done:

* The ability to produce and send anti-msgs
* The ability to rollback when either events from the past are recv’d or anti-msgs are scheduled/recv’d

The following subsections will discuss these tasks.

## Anti-msg Management

Anti-msgs will be events that are send and scheduled to processes to cancel scheduled events or rollback events executed. Now, to do this, there are two tasks to complete:

1. How to determine the where, what, and when
   1. Where to send?
   2. What event-action anti-msg associated with?
   3. When is the event?
2. How to send anti-msgs
   1. When event is rollbacked, how to schedule anti-msgs?

The solutions to the first two tasks are to associate event scheduling relative to event-action class itself instead of sim-exec. I.e., if an event schedules another event, it will use event-action’s event scheduler instead of sim-exec’s. What the event-action will do differently than sim-exec is during scheduling, the event-action will save the event scheduled (process sent too, time of event, and event random identifier) to a list then schedule the event with sim-exec. Doing this will allow the event-action (EA) when rollbacked over to send anti-msgs via a send anti-msg method (i.e., anti-msgs are sent when EAs are rollback on).

The random event identifier is used to consider simultaneous events. Every new EA created on the process will be associated with random ID. Therefore, when anti-msgs are sent and events with same time are found, the anti-msg knows exactly what event to remove or rollback. To do this, I am assuming for two events to have same random number and same event time is extremely low. Also, random numbers will be created by each process where each process will have different seed (decreasing the probability even further).

The following is pseudocode for scheduling events and the destructor which sends events:

*EA::ScheduleEvent(event time (et), new EA (ea), process):*

*Save scheduled event information to anti-msg list (ea’s event id, process, and et)*

*SimExec::ScheduleEvent(et, ea, process)*

*End ScheduleEvent*

*EA::SendAntiMsg ():*

*Foreach anti-msg in anti-msg list:*

*Schedule anti-msg - SimExec::scheduleEvent(anti-msg)*

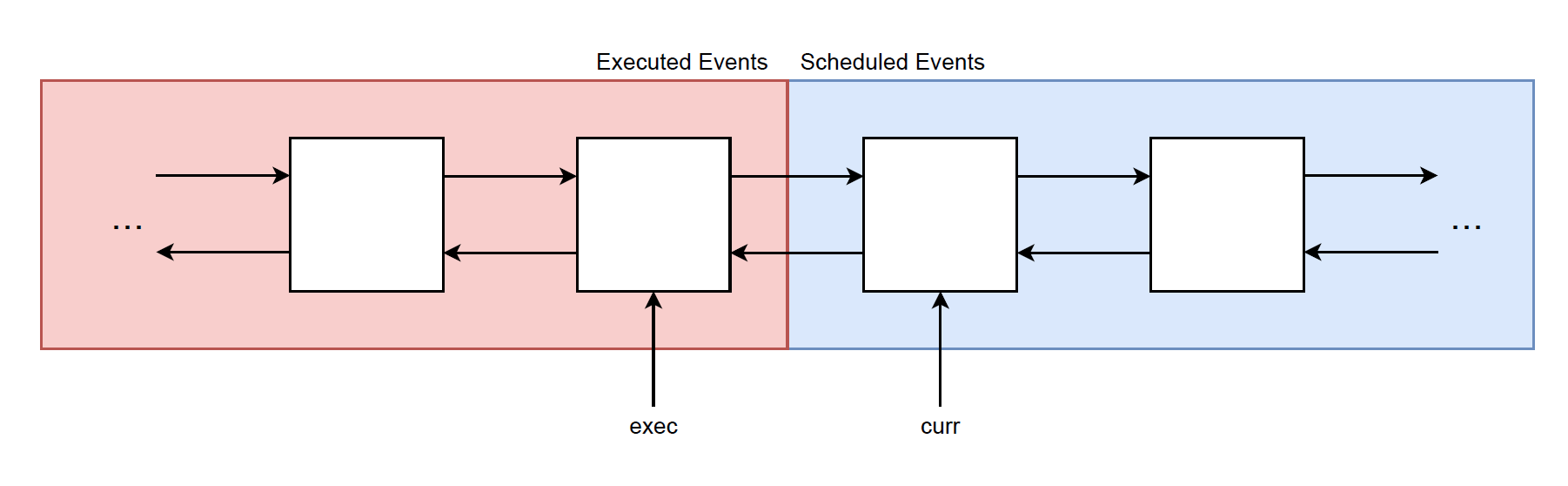
*Remove anti-msg from list*

*END SendAntiMsg*

## Event-Set Rollback Mechanism

With the design for producing and sending anti-msgs is done, the next task is to determine what to do with anti-msgs when scheduled and how-to rollback on positive events. To do these tasks, a classical DES event-set will be modified to do all the logic for rolling back, calling anti-msgs senders, and scheduling both anti-msgs and new events.

The event-set (ES) is a doubly linked-list with two sets contained inside list: executed events and scheduled events. Figure below illustrates linked list used. The executed set will contain events that were executed by sim-exec. The scheduled set are events that are scheduled to be executed by sim-exec. As shown in the figure below, there are two points: exec and curr. exec points to the previously executed event, and curr points to the next event to execute. When curr gets executed, it will move to the right, moving to the next event in the ES, and the same time, exec will move to the right to signify the execution of curr. Using this ES configuration, events can be rolled back by moving exec and curr to the left until event to rollback has been reached.



For a rollback to occur two things can happen. First, an event from the past has arrived (timestamp less than current simulation time). Second, an anti-msg canceling an executed event is scheduled. In both scenarios, rollbacks will occur, and during the rollback, possible anti-msgs can be sent from rolling over executed events. Because of this behavior, the logic behind rollback will be the event-set scheduler.

The scheduler will have two sections: when anti-msgs are scheduled and when events are scheduled. In the first section, the scheduler will have to determine where the event the anti-msg is associated is, and if event cannot be find, where to schedule it into the set so to wait for the event to arrive, and If the schedule finds the event, then it will eliminate event from list, and if event is in executed domain, then rollback to previous event to the event just eliminated. In the second section, the scheduler will have to determine where to place the event and if this event has an associated anti-msg waiting for it to arrive. If the event is scheduled in the past, a rollback must occur.

The pseudocode for the scheduler is below. The scheduler first determines whether the event-action to schedule is an anti-msg or not. An anti-msg is a special type of event-action. What makes it special is that it has an event class ID of zero. With this, the scheduler can determine whether it is an anti-msg. Based on the decision, either the first section (anti-msg scheduling) will occur or second section will occur.

First section occurs. After determining that new event (ea) is anti-msg, the scheduler will determine if ea time (et) is greater than curr et, less than exec et, or equals curr et or exec et. If none of those tests are true, then scheduler will place anti-msg between curr and exec and repoint exec at anti-msg since event associated with anti-msg has not arrived. The following will explain procedures done for each test case if true:

* If et is greater than curr et, then search scheduled events starting from curr to scheduled event with greater timestamp than et or end of list. If event associated too ea (same time and event identifier) is found, delete event from list. Else (event is not found), add anti-msg to list so to wait for event to arrive.
* If et is less than exec et, then search executed events starting from exec to executed event with smaller timestamp than et or end of list. If event found, remove event from list, rollback all events starting from exec too event found (send anti-msgs for each event rolled over), point exec at prier, prier event to found event (if exists) and point curr to prier event to found event (if exists). Else (event no found), schedule anti-msg to wait for event to arrive.
* If et equals exec et or curr et, search scheduled events with same time as et. If found, eliminate event. Else, search executed events with same time as et. If found in executed events, rollback to prier event. Else, add anti-msg between exec and curr and repoint exec at anti-msg so to wait for event to occur. This test is used if there are simultaneous events between exec and curr.

Second section occurs. After determining that new event is not anti-msg, the scheduler will determine if et greater than exec et, less than curr et, or equals curr et or exec et. If none are true, then schedule between exec and curr and repoint curr to new event. The following are the procedures for each test case:

* If et is less than exec et, then search executed events from exec to event with smaller timestamp than et or end of list. While searching, if anti-msg found associated to ea, stop search remove anti-msg from list, and do not schedule ea. Once searched ended (no anti-msg), add new event to list and rollback to new event.
* If et is greater than curr, then search scheduled events from curr to event with greater timestamp than et or end of list. While searching, if anti-msg found associated to new event, stop search remove anti-msg from list. Once searched ended (no anti-msg found), add event to scheduled domain.
* If et equals curr et or exec et, then search all executed events with same time stamp for anti-msgs. While searching, if anti-msg found, stop search, and remove anti-msg and do not schedule new event. If no anti-msgs found, search all scheduled events with same timestamp for anti-msgs. If found, remove anti-msg and do not schedule new event. Else, add new even between curr and exec and repoint curr to new event.

*ES::Scheduler(new event time (et), new EA (ea)):*

*If ea is anti-msg (ea->ID == 0):*

*if et is greater than curr’s time:*

*search from curr to event time <= et OR end of list for associated event to ea*

*// will be searching from left to right starting from curr*

*if found:*

*eliminate event from list and repoint previous and next events*

*else:*

*add anti-msg to list to wait for event to arrive*

*else if et is less than exec’s time:*

*search from exec to event time >= et OR end of list for associated event to ea*

*// will be searching from right to left starting at exec*

*If found:*

*Eliminate event from list*

*repoint previous and next events*

*rollback to previous event*

*// where each event rolled over will send anti-msgs*

*Else:*

*Add anti-msg to list to wait for event to arrive*

*else if et is equal to either curr’s time or exec time:*

*search curr from curr to all events times == et for associated events too ea*

*// left to right starting at curr*

*if found:*

*eliminate event from list and repoint events*

*else:*

*search exec from exec to all event times == et for associated events to ea*

*// right to left starting at exec*

*If found:*

*Eliminate event from list*

*Repoint events*

*Rollback to previous event*

*Else:*

*Add anti-msg in-between curr and exec*

*Point exec at new anti-msg*

*else: // schedule (no event w/anti-msg time found)*

*add anti-msg to between exec and curr*

*point exec at anti-msg*

*else: // not anti-msg*

*if et less than exec time:*

*search from exec to event time >= t*

*if anti-msg is found during search:*

*eliminate anti-msg and do not schedule event*

*else:*

*add event to list*

*rollback to event*

*else if et greater than curr time:*

*search from curr to event time <= t*

*if anti-msg found during search:*

*eliminate anti-msg and do not schedule event*

*else:*

*add event to list*

*else if et equals either curr time and/or exec time:*

*search exec from exec to event time == et for anti-msgs associated to ea*

*if found:*

*eliminate anti-msg and do not schedule ea*

*else:*

*search curr from curr too event time == et for anti-msgs to ea*

*if found:*

*eliminate anti-msg and do not schedule ea*

*else:*

*add ea to list*

*else: // add to start of schedule event set*

*add ea between curr and exec*

*curr points to ea*

*exec points to previous ea*

*END scheduler*

Due to the behavior of the scheduler, there is the possibility that curr is pointing to anti-msg for which anti-msg are not executed and must be in timestamped order. Therefore, to account for this possibility, new ES is empty method was created. The empty method checks if there are executable events in scheduled set, if there are then return false (not empty), and else, return true (is empty). However, if there is an anti-msg at curr, and possibility more adject in list in scheduled domain, then ES is empty will only return false if it can guarantee there is at least one executable event beyond any anti-msgs in scheduled domain. To do this, is empty will send a probe starting at curr to first executable event (if exists). If probe comes back not pointing to null, then executable event was found and curr will be pointed at events and exec will be pointed at previous event and return false. Else (probe came back null), then return false (either curr is null or anti-msgs were found and no executable event). Pseudocode below:

*ES::isEmpty():*

probe *= curr*

*While* probe *AND* probe *->ea->id == 0:*

probe *=* probe *->next*

*if tmp:*

*curr =* probe

*exec =* probe *->previous*

*return false*

*else:*

*return true*

*END scheduler*

# Results

Based on the design and pseudocode in the previous section, I created an application to test the optimistic algorithm based on task 2 and 3 descriptions in HW 5 System Requirements document. The application will run with n-events from each processor, and where each event will never be destroyed during the simulation. Therefore, resulting in a non-terminating simulation due to events. Also, to see desired optimistic behaviors (rollbacks), the events will be scheduled with exponential distribution with mean time delay (td) and each event execution will result in a random delay (sleep) on processor where the sleep duration is calculated using uniform distribution from 0 to max wait time (tw). Also, each process max wait time will be a fraction of global max wait time based on communication rank, resulting in a process being on average lower than the rest. The fraction is max wait time divided by communication rank plus one.

Using this application, three tests conducted with differing number of processes (LPs), td’s, max wait times, and termination times. The tests results can be found In ApplicationResults.xlsx. The excel spreadsheet will show test parameters, output statistics per process (number of rollbacks, number of events rolled back, and average number of events rolled back), the raw data, the data for each processor, and mark when rollback occurred.

## Observations

Based on the data collected, there are a few observations that can be made.

First observation, impact of parameter selection. It was noted that the selection of tw did not drastically affect the outcome of the results. The only difference that did occur was how long the simulation took relative to wall clock time. The other parameters, td and number of events, had an impact where the number of event added would increase simulation run time relative to wall-clock and would increase the likely hood of rollback since more events are the system, and where td would for large value decrease simulation run time relative wall-clock and for smaller values it would increase the likely hood of rollback since events were close together (i.e., takes longer to get to termination time).

Second observation, observing rollback behavior. Table 1 is a snippet of test 1. As seen in the table, processor 1 executed an event 5409 at 1.36477, but then received new event from processor 0 causing rollback (highlighted areas). After rolling back, processor executed new events and executed previously executed events that were rolled over (e.g., 5409 re-executed post rollback). With this observation, I can say with confidence that my optimistic algorithm is working correctly.

Table . Test 1 Snippet

|  |  |  |
| --- | --- | --- |
| Process | Event ID | Timestamp |
| 0 | 1164 | 0.325927 |
| 1 | 5409 | 1.36477 |
| 0 | 22454 | 0.617311 |
| 1 | 26285 | 0.970256 |
| 1 | 31891 | 0.442696 |
| 1 | 26285 | 0.970256 |
| 0 | 2023 | 3.566528 |
| 1 | 5409 | 1.36477 |