Distributed Simulation

Optimistic Algorithm

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

Please read over Assignment 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 three 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 does 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) 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. Therefore, 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*

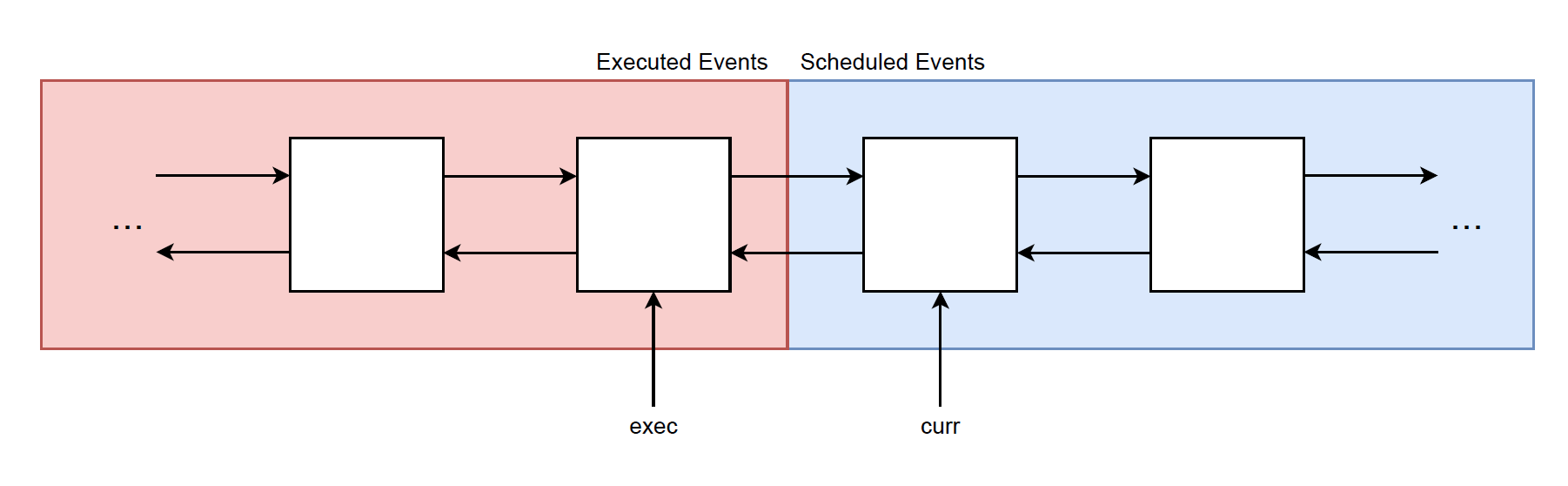
*END SendAntiMsg*

## 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 (exec) and scheduled events. 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. With these two domains within the ES, previous events can be rolled back onto.

Figure below illustrates linked list used. As shown in the figure, 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.



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 it cannot find it, where to schedule it into the set so to wait for the event to arrive. If the schedule does find 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 a 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 (true) or second section will occur (false).

*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 and/or exec time:*

*if et equals both curr and exec times:*

*search 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 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 to list to wait for event to arrive*

*Else if et equals exec time:*

*search 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 to list to wait for event to arrive*

*Else: // et == curr time*

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

*// left to right starting at curr*

*If found:*

*Eliminate event from list*

*Repoint events*

*Else:*

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

*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:*

*if et equals both curr and exec:*

*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 if et equals 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:*

*add ea to list*

*rollback to ea*

*else: // et equals curr*

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

# Results

Based on the design and pseudocode in the previous section, I created an application that traces the rollbacks