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)*

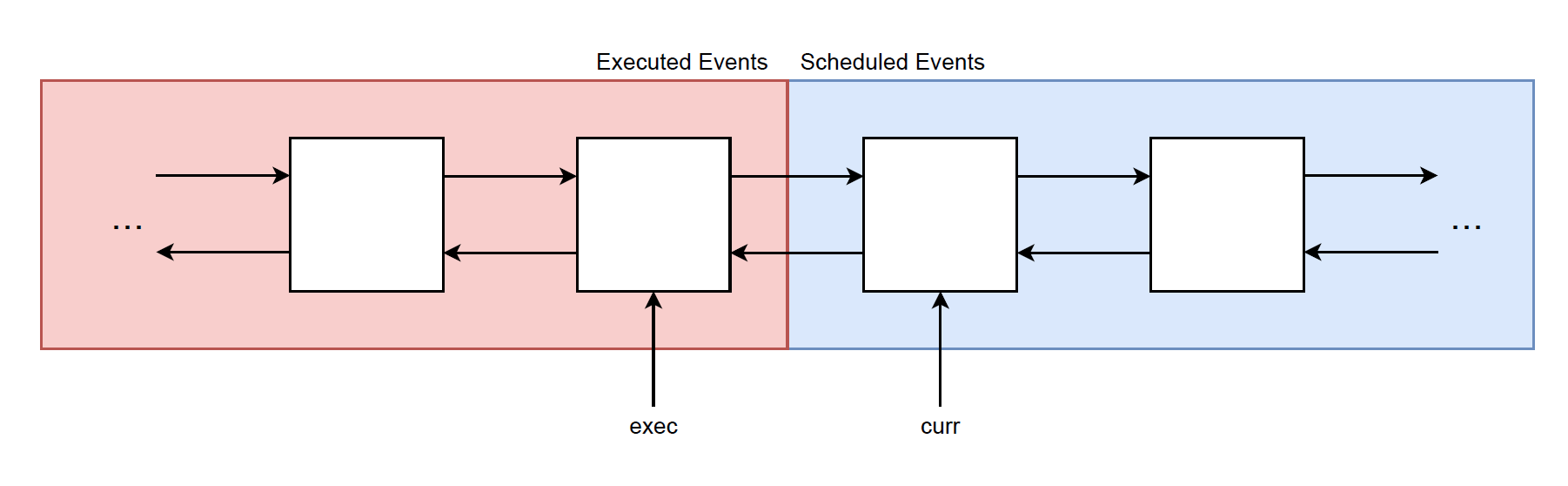
*Remove anti-msg from list*

*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. 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.



# Results