Ada 95 for a Distributed Simulation System

Helge Hagenauer and Werner Pohlmann

Institut für Computerwissenschaften und Systemanalyse
Universität Salzburg
Jakob-Haringer-Straße 2
A-5020 Salzburg
Austria
hagenau@cosy.sbg.ac.at, pohlmann@cosy.sbg.ac.at

Abstract. In the distributed discrete event simulation area, Jefferson’s time warp algorithm initiated a lot of research and practical work. We proposed a generalisation, split queue time warp, that allows lazy message reception and thus may reduce rollback frequency. The present paper contains a brief description of our algorithm and then describes an implementation that uses Ada 95 and its capabilities for distributed programming.

1 Introduction

In distributed discrete event simulation (for an introduction see [10], [5] and [4]), the traditional concept of a central simulation clock and its event-list implementation are replaced by a very different approach. One now views a system under study as composed of loosely coupled “physical processes”, PPs, and simulates them by “logical processes”, LPs, that interact exclusively by sending and receiving timestamped messages. The problem then is to maintain causal correctness. There are two main solutions, the “conservative” and the “optimistic” one, and the second, well-known as “time warp”, [9], is our subject here. Time warp combines a no-wait strategy with an error detection and recovery scheme. LPs never wait for possible further messages but proceed on the basis of whatever has been received so far. This “optimistic” behaviour runs the risk of getting ahead of its input and thus producing errors. So if and when a LP receives input that should have been considered earlier, it must rollback to a former state and cancel its unjustified interim output (thus possibly inducing rollbacks elsewhere). Clearly, rollbacks and especially cascades of them form a major cost and reduce the advantage of having work done in parallel.

In standard time warp, henceforth called TW, all messages sent to a LP are inserted into and read from a single input queue that is maintained in ascending timestamp order, and corrective actions are triggered by “time faults”, i.e. by the arrival of a message whose timestamp is smaller than the receiver’s already achieved temporal level. This strategy is safe but possibly overcautious since the
causality relation is covered by temporal order but usually much sparser than it: Many real-world processes are "active" in the sense of choosing, in relation to current state, what kind of information is relevant for their next move and what can be postponed.

Example: A typical case is a LP that models synchronisation of model entities; consider e.g. taxi-cabs and customers arriving at a taxi-stand. If there already is a customer present, the next output of the taxi-stand LP depends on the availability of a free taxi and not on the arrival of more customers. Consequently, in this state, the LP may ignore incoming messages denoting customer arrivals and instead look for a returning taxi.

Processing messages in strict timestamp order has two drawbacks: First, the LP has to copy currently unwanted information from the input queue and temporarily save it in its internal state. Second, the premature reading and copying of not-yet-needed messages inflates the past of the LP with event messages that, in time warp, may be erroneous or still subject to time faults; consequently, the LP runs a superfluously increased risk of rollbacks.

We therefore proposed to furnish the LPs with several input queues each of which corresponds to a message type and is read only when necessary for progress, i.e. message processing is now "lazy" or "by need". Figure 1 illustrates this for the taxi-stand example. The basic no-wait/error-recovery approach of time warp can be generalised to work with the new arrangement; the need for rollbacks is now discovered by comparing changes in the input queues against how far the queues were already read. We call the new algorithm split queue time warp or SQTW ([6]).

```plaintext
LOOP
  IF customer_is_present
  THEN
    get(message, taxi_channel);   -- seize taxi
    local_time := max(message.timestamp, local_time);  
    send((taxi, local_time + some_travel_delay), somewhere);
    customer_is_present := false;
  ELSE
    get(message, customer_channel);  -- next customer
    local_time := max (message.timestamp, local_time);  
    customer_is_present := true;
  END IF;
END LOOP;
```

Fig. 1. Taxi-stand model for SQTW

We first tried out our idea in a pseudo-parallel one-processor prototype written in Ada 83 [7] and found that SQTW can indeed lead to a decrease in rollback numbers. In the present paper we describe a new and more adequate implemen-