Time Management in Active Networks

Craig Lee, James Stepanek, Cauligi Raghavendra† and Kirstie Bellman

Computer Systems Research Department
The Aerospace Corporation, P.O. Box 92957
El Segundo, CA 90009
{lee|stepanek|bellman}@aero.org

†Dept. of Electrical Engineering Systems
University of Southern California
Los Angeles, CA 90089
raghu@halcyon.usc.edu

Abstract

This stuff is pretty abstract.

1 Introduction

Time Management is a central issue in parallel and distributed simulation [4]. Entities that are being simulated in parallel must interact by some form of message-passing. This means the *causality* between different messages could be violated unless some type of message *ordering* is enforced. While different simulations can require different ordering semantics, message *time stamps* are commonly used to observe causality. Hence, Time Management refers to the management of simulated time at each host, its progression, and the order in which messages can be delivered to their destinations. This serves as a cornerstone in the High-Level Architecture (HLA) for distributed simluations [11] defined by the Defense Modeling and Simulation Office.

The performance and efficiency of time management can clearly have a great impact on the overall behavior of a distributed simulation. In a distributed environment, the performance and efficiency of time management greatly depends on that of network communication. Simulation is generally, by its nature, tightly coupled but typical distributed simulation tools use simple, point-to-point network communication. This means that end-hosts will typically need to communicate with small, frequent messages but in an environment that can make communication latency a dominant factor in the simulation's performance. Specifically for Time Management, simple, point-to-point communication means that ordering semantics must be implemented using multistage synchronization algorithms that require multiple sets of messages that span the entire network of end-hosts.

Active networks offer a unique way to address these communication issues for time management and distributed simluations. Active networks allow applications to maintain and manipulate state on specific routers on a per-flow basis [14]. **(More recent survey article?)** This allows active routers to be "smart" and greatly enhance the functionality of communication networks. For Time Management, active networks could be used to reduce or eliminate the need for point-to-point communication and replace it with reductions that are done on-the-fly in the network, i.e., at the routers. This could greatly reduce both the number of messages and latency required to support Time Management.

Hence, this this paper examines the issue of supporting Time Management in an active network. We begin with a discussion of Time Management followed by a discussion of how active networks have been used in other aspects of distributed simulation. We then identify the fundamental issues in Time Management designs and how this relates to the Distributed Termination Detection problem. Three designs are subsequently presented that take different fundamental approaches and we analyze their complexity.

2 Time Management

2.1 Fundamentals

In the real world, events are ordered by time and their observability is determined by many factors, such as line-of-sight, signal propagation, etc. Under simulation, the typical goal is to faithfully reproduce the important events and their observability. In a simulated world, however, events and their observability can be affected by the simulation machinery. Consider Figure 1. This illustrates a scenario where a simulated Shooter fires on a simulated Target which is observed by a simulated Observer. The dotted lines indicate delivery of events. The Fire Event necessarily preceeds the Destroy Event. Due to irregular delays, possibly in a computer or network, the Fire and Destroy Events are delivered to the Observer out-of-order. (Other interesting causality scenarios are described in [4].)

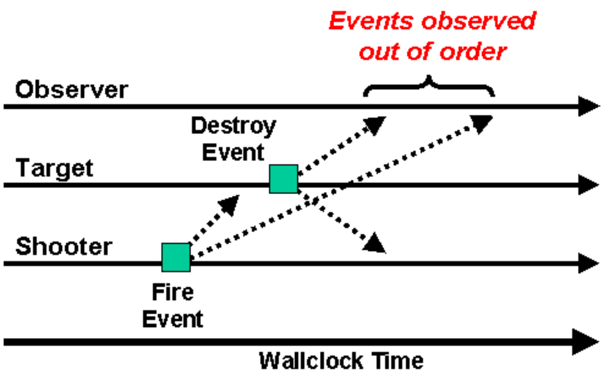

Figure 1. A Temporal Anomaly (after Fujimoto).

Clearly, simulation must maintain causality to ensure correctness. In the real world, causality is constrained by *time*. An event at time t cannot be caused by an event at time $t + \delta t$ (for positive δt). In simulated worlds, and especially in parallel and distributed simulation, causality must be constrained by knowing the *simulated time*. Simulations run in simulated time which typically does not necessarily relate to wallclock time. In a parallel and distributed simulation, *temporal anomalies*, such as illustrated in Figure 1, can be caused by simulated time progressing at different rates on different hosts, and messages containing event information being delivered in different orders. Since parallel simulations can only interact by messages (regardless of how they are implemented), *message-ordering services* must be used to prevent temporal anomalies.

Three message-ordering services are:

- Causal Order
- Causal and Total Order, and
- Time Stamp Order.

A Causal Order message service ensures that if "E_1 *happens before* E_2", that all hosts receiving both events will receive E_1 before E_2. Causal Order, however, is not sufficient to prevent all temporal anomalies. If two events that are not causally related, i.e., one does not "cause" the other, they can be delivered to different hosts in different orders. The shared, simulated world at these two hosts, however, may diverge if the order of these events, e.g., which is seen first, is important. In this case, a Causal and Total Order delivery service would have to be used that ensures that all hosts see all events in the same order. Even Causal and Total Order, however, is not sufficient to prevent all temporal anomalies. Simulation semantics that are specifically associated with simulated time or timing, e.g., "Begin the attack at 0500.", may fail since many causal and total orderings may be possible but only one order may be correct. For this reason, a Time Stamp Order delivery service is used to time stamp every event and message and deliver each message in nondecreasing time stamp order.

In a Time Stamp Order delivery service, simulated time becomes a "virtual wallclock time" for a distributed virtual environment. Hence, the delivery service must accomplish some minimal *time synchronization* to determine when messages can be delivered. Rather than enforce a strict synchronization across all hosts in a parallel simulations, the *lower bound time stamp* across all hosts is sufficient to know when delivery can be done. If n hosts are at simulated times $t_1, \ldots, t_n$, then it is safe to deliver messages with a time stamp

$$t_{msg} \le t_{min} = min(\, t_1,\, \ldots,\, t_n\,)$$

since no other messages could be generated with a earlier time stamp.

Parallel and distributed simulations, however, have an additional complication. Messages sent between hosts have a time stamp and take some non-negligible time to be delivered. In a heterogeneous, networked environment, it is possible that t_{min} could be computed and then a message with $t_{msg} < t_{min}$ arrives. To prevent this problem, the *Lower Bound Time Stamp (LBTS)* computation is defined over all hosts *and in transit messages*:

$$LBTS = min(\, t_1,\, \ldots,\, t_n,\, in\, transit\, messages\,).$$

The inclusion of in transit messages is an important factor in the design and implementation of LBTS algorithms.

At this point, we note that the LBTS and GVT (Global Virtual Time) computations used in distributed simulations are similar but not identical [12, 13]. LBTS is used in conservative simulations where the *look-ahead* determines the degree of concurrency. Computing LBTS is essential to allow the simulation to progress. GVT is used in optimistic simulations where events are consumed optimistically and *roll-backs* occur if an event arrives in the past. GVT is used to trigger *fossil collection*, i.e., garbage collection of old messages. Hence, LBTS should be *eager* and computed more frequently to enable the distributed simulation to progress, whereas GVT can be *lazy* and computed less frequently as long as the computation does not exhaust memory. A more detailed discussion of these differences is not necessary here and the interested reader is referred to [4].

2.2 LBTS Algorithms and Implementations

The computation and use of the LBTS value is the basis of *Time Management* in many parallel simulation systems including the High Level Architecture (HLA) [3, 11] of the Defense Modeling and Simulation Office. The LBTS computation is essentially a reduction (minimum) over a bounded but variable number of time stamps. The number of time stamps is variable since it includes end-host time stamps and any in-transit messages at the time of the computation. The fact that the number of time stamps is variable introduces a termination problem. At the start of an LBTS computation, the number of in-transit messages is unknown, in addition to where they are and their destinations. Hence, a static communication structure cannot be used to easily determine when the LBTS has been computed. This means that LBTS algorithms must not only compute a reduction, they must also detect in-transit messages, not only to include them in the reduction but also to know when the computation is complete.

Reductions are a well-known problem in parallel computing. Reductions can be computed using a centralized controller with $O(n)$ behavior. This can be acceptable for small n but can become a performance bottleneck for larger n. Pair-wise or k-way linear reductions, however, can be used to build n-way parallel reductions using, for example, *butterfly, tree*, or *skip-ring* topologies that have $O(log_k n)$ stages [6, 7]. The efficient and reliable detection of in-transit messages poses a larger challenge.

A simple way to detect in-transit messages is to rely on the maximum message latency l between any host pair in the simulation. If t is the current host clock time and incoming messages are accumulated in a queue, then all messages with a time stamp less than $t + l$ must include all transit messages. Of course, this assumes that host clocks are perfectly synchronized and that the latency l is bounded and fixed. To account for clock skew, a maximum clock offset could be added. In a heterogeneous, network environment, latencies may be quite variable and require that the l bound be set correspondingly "loose". Hence, relying on message latency to accomplish time management requires a trade-off between introducing higher than necessary message delivery latencies versus risking unreliable in-transit message detection.

A reliable method for in-transit message detection uses a *coloring scheme* [5, 10]. All hosts are initially the same color, say red, and all messages sent are tagged with the sender's current color. When a host starts a new LBTS computation, it changes color, to say green. Hence, when a green host receives a red message, that message was in-transit at the time LBTS was started. Colors need not be limited to red and green. Since it can be advantageous to allow another LBTS computation to start before the previous one has completed, implementations can allow upto k "colors" to be in simultaneous use.

Termination is detected by the use of *message counters.* Each host maintains a message *sent counter* and a *received counter*. (In practice, a single counter can be used that records the difference between the number of sent and received counters.) When an LBTS computation is started, a separate sum reduction is done across all send and receive counters. When the total number of messages sent equals the total number of messages received (or if the difference counters sum to zero), then all in-transit messages have been seen. Of course, if there are one or more in-transit messages at the start of LBTS, then the counter sums will not be equal. Hence, when a host detects an in-transit message, it must increment its receive counter, and include the new receive count and the time stamp on the message into the reductions for both the LBTS and termination detection.

3 Related Work

A number of prototypes, tools, and testbeds for active network research have been developed [14, 8]. Rather than survey all of these prototypes, tools and testbeds and discuss their low-level details here, we instead discuss how active networks have been applied to other aspects of distributed simulation.

Active networks have been applied to *interest management* [15]. In the HLA, simulated entities interact through a *routing space* defined over n dimensions. Subregions of the routing space can be defined based on subranges in each of the n dimensions. These subregions can be *update regions* or *subscription regions*. Simulated entities can subscribe to regions they are interested in and publish, or update, information in other regions. Hence, in a network environment, the only data that is transmitted is that with one or more subscribers, thereby greatly reducing unnecessary traffic. The actual routing of data is determined by the *hyper-box intersection* of the n-dimensional update and subscribe regions.

In current work following [15], interest-based routing is implemented by using an *Active Filter Signaling Protocol* (AFSP). **(References needed for this stuff.)** End-hosts use AFSP to signal their interests within a common multicast group. At an active router within this multicast group, a signaling packet commands a *Filter Manager* to create or delete one or more filters. As a signaling packet progress through the multicast group, the *filter spec* it carries can change based on previous signaling received from other end-hosts. The installed filters pass or drop packets based on their region tags. We note that hyper-box intersections can be very complex, especially for routing spaces with a large n. Multiple hyper-boxes, however, can be subsumed into one, larger hyper-box. This allows filter complexity to be reduced at the cost of allowing some unnecessary data to be transmitted.

A very different application of active networks to simulation is reported in [1, 2]. The *Active Virtual Network Management Protocol* (AVNMP) uses a form of optimistic simulation to make future, predicted state information available to network management algorithms. Predictive state information can be used, for example, to improve the hand-off of mobile, wireless communication since it can be set-up ahead of time rather than being a reactive event. As another example, predictive load information could be used to improve Quality of Service.

AVNMP wraps each *physical application process* in a *logical process* running in a router or switch. Besides handling the *real messages* for the application, a logical process also handles *virtual messages* which contain future event information and have a time stamp greater than the current time. Logical processes can use the future information to make predictive choices while maintaining a running comparison with the real information. To generate predictive information, a *driving process* will inject *streptichron* active packets that refine a prediction as it travels through the network. To evaluate the accuracy of predictions, the *global virtual time* is computed by *GVT* active packets. **(Exact details pending discussion with Steve Bush.)**

4 Design Properties

As mentioned above, the LBTS computation is essentially a *min* reduction over a bounded but variable number of time stamps. The reduction means that LBTS (and GVT) involve a synchronization that produces some aspect of global state concerning the simulation. The number of time stamps is variable since in-transit messages must be included in the reduction and their location is not known ahead of time. Hence, LBTS is another instance of the *Distributed Termination Detection* (DTD) problem where the hard problems concern the reliable and efficient detection of termination (completion) and the reduction is comparatively incidental and, as usual, could be any commutative operator. DTD as a problem was identified very early in the history of distributed computing but, due to its fundamental nature, has since been widely studied in many contexts. We cite this one paper [9] for this entire topic area.

While LBTS is essentially a DTD problem, there are some important differences. A distributed system is usually defined as a set of n processes P connected by a set of communication channels E with the following assumptions:

- There is no shared memory.
- There is no shared clock.
- Communication takes an arbitrary, but finite time.

LBTS meets these criteria. A distributed computation is modeled as follows:

- There is a *basic computation* that produces *basic messages*, i.e., those not involved in DTD.
- A process in the basic computation is either *active* or *passive*.
- An active process may become passive at any time.
- Only an active process can send a basic message.
- A passive process can only become active by receiving a basic message.
- A computation has terminated if all processes are passive and there are no in-transit messages.

The issue here is that a passive process may or may not have terminated. That is why basic messages must be accounted for. In LBTS, "becoming passive" is similar to "starting the i-th LBTS computation". It represents a point in time when some global knowledge of the computation must be ascertained that depends on all other processes and in-transit messages. The important distinction here is that, with regards to LBTS, processes are not passive or active. Rather than computing a boolean (terminated or not terminated) with a boolean operator, LBTS computes a reduction with a non-boolean operator.

Hence, we can use the same properties to evaluate LBTS as well as DTD algorithms:

- Algorithm type.
- Topology.
- Symmetry.
- Process knowledge.
- Communication protocol.
- Message arrival order.
- Message optimality.
- Fault Tolerance.

mcast breaking counter algos.

Active Network Aware TM Intermediate active nodes, or routers, participate in algorithms Transient detection in the network Minimum time reduction in the network Topology aware explicit tree structure in the network Should improve scaling, efficiency, and performance of time management

In classical DTD, the edges are p2p. In AN case, active routers become a processor. But things are not exactly the same, even then. An AR doesn't initiate the computation, only the end-hosts do. The ARs don't care about the outcome; only the end-hosts do. The ARs don't need to know about the outcome; though they may.

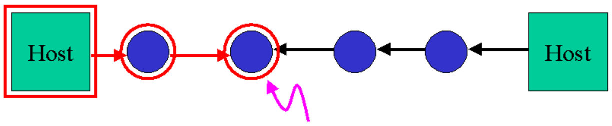

Figure 2. Detection of in-transit messages.

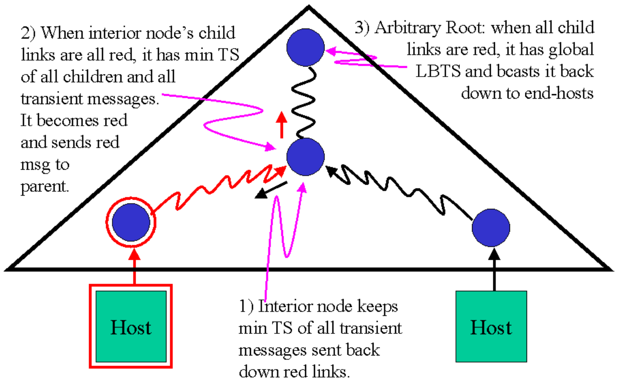

Figure 3. A Tree-Connected Federation.

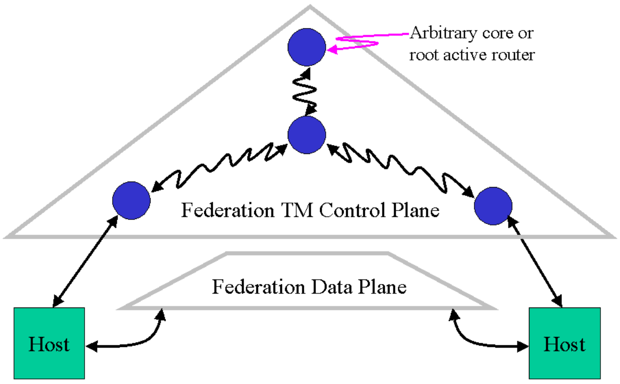

Figure 4. A Tree-Connected Federation in Two Planes.

5 Three Designs

How can active networks be used to support time management? As noted above, active networks allow an application to maintain and manipulate state on specific routers on a per-flow basis.

General TM Criteria requirements: scaling properties overhead and efficiency calculation throughput calculation latency

5.1 No Distinguished Node

Combined Control and Data Planes but no root node.

5.2 Distinguished Interior Root Node

Combined Control and Data Planes but with a root node.

Basic Problem: Detecting Transient Messages and including them in the LBTS Computation

Tree-connected AN Federation

Notes When one node enters LBTS, all nodes must enter LBTS In Fujimoto, when a node receives a Butterfly LBTS msg, presumably this tells it to enter LBTS If multiple nodes start LBTS, the computations are merged In AN Tree, when one node enters LBTS, it should bcast to all other nodes to put them in LBTS This could be part of the function of a red message; when a router sees this message, it knows to bcast it back down any black links. If multiple hosts bcast Enter LBTS, the AN routers should merge the bcasts into one If a router gets a red msg from its parent, it knows to merge it with any red msgs from child links.

Assumptions Reliable Delivery Static Routing Relies on known, static tree topology Non-overtaking messages Red messages must never overtake transient messages Transient message can never come from a red child link Can any of these be relaxed?

Advantages over End-Host LBTS (as described in Fujimoto/Hoare paper) Detection of transient messages is short-circuited by detection in the interior of the network Faster and consumes less BW Tree fan-in/fan-out consumes less BW than butterfly barrier Butterfly barrier sends nlogn messages between end-hosts (across entire network) AN Tree sends fewer messages and has shorter paths

Issues Will this be a significant benefit? Network simulations to analytically compare performance? Current HLA/RTI TM implementations may be slow but how much of this is due to network latency or congestion vs. other subtle details in a real-world RTI implementation or just dumb/nave implementations? Can any of the assumptions be relaxed? These assumptions might not be sustainable in a real, deployed environment.

5.3 Separate Control and Data Planes

Basic Problem: Detecting Transient Messages and including them in the LBTS Computation

AN Federation w/ TM Tree Control Plane

Notes All End-hosts, control plane routers, and data messages have two states: Green = normal operation Red = in LBTS computation NOTE: This is opposite of the Fujimoto paper but makes more sense to me! Each End-host maintains Send/Recv counters for data messages Same as Fujimoto paper Can be simplified to one Send/Recv difference counter There are four different TM control messages, in addition to data messages: startLBTSmsg: Start the LBTS computation myLBTSmsg: Report my LBTS value to parent Only flows up towards root corrLBTSmsg:

Correct the LBTS due to the arrival of transient messages Only flows up towards root trueLBTSmsg: Report the true LBTS value to all children and end-hosts Only flows down towards end-hosts

Advantages over End-Host LBTS (as described in Fujimoto/Hoare paper) Tree fan-in/fan-out consumes less BW than butterfly barrier Butterfly barrier sends nlogn messages between end-hosts (across entire network) AN Tree sends fewer messages and has shorter paths End-hosts must detect transient messages but TM Control Plane is separate from Data Plane Data messages route at line-speed

Issues Will this be a significant benefit? Network simulations to analytically compare performance? Current HLA/RTI TM implementations may be slow but how much of this is due to network latency or congestion vs. other subtle details in a real-world RTI implementation or just dumb/nave implementations? Can any of the assumptions be relaxed? These assumptions might not be sustainable in a real, deployed environment.

These operational semantics do not indicate any limit on the number of simultaneously active LBTS computations. In practice, however, this would be limited to some finite k.

5.4 Performance Analysis

6 Outstanding Issues

Lots.

References

[1] S. Bush. Active virtual network management protocol. In *13th Workshop on Parallel and Distributed Simulation*, May 1999.

[2] S. Bush and A. Kulkarni. *Active Networks and Active Network Management: A Proactive Management Framework*. Kluwer Academic/Plenum Publishers, March 2001.

[3] C. Carothers, R. Fujimoto, R. Weatherly, and A. Wilson. Design and implementation of HLA time management in the RTI version f.0. In *Winter Simulation Conference*, 1997.

[4] R. Fujimoto. *Parallel and Distributed Simulations Systems*. Wiley, 2000.

[5] R. Fujimoto and P. Hoare. HLA RTI performance in high speed LAN environments. 1998.

[6] D. Hensgen, R. Finkel, and U. Manber. Two algorithms for barrier synchronization. *International Journal of Parallel Processing*, 17(1):1–17, 1988.

[7] C. Lee. Barrier synchronization over multistage interconnection networks. In *2nd IEEE Symp. on Parallel and Dist. Processing*, pages 130–133, December 1990.

[8] C. Lee. On active grid middleware. *Second Workshop on Active Middleware Services*, August 2000.

[9] J. Matocha and T. Camp. A taxonomy of distributed termination detection algorithms. *J. of Systems and Software*, 43(3):207–221, 1998.


```
/*  Sent or recvd startLBTSmsg(i):  */
if  (myColor != i)  {
    myColor = i;
    send myLBTSmsg(i) to parent
      in Control Plane;
}


/*  Recvd a datamsg(msgColor):  */
if  (myColor > msgColor)  {

    /* intransit message */
    send corrLBTSmsg(myColor+1);

}  else if  (msgColor > myColor)  {

    /* Somebody has started i=msgColor
       but I haven't gotten
       startLBTSmsg(i) yet;
       enter LBTS(i) anyway.
    */
    myColor = i;
    send myLBTSmsg(i);

}
normal message processing;


/*  Recvd trueLBTSmsg(i):  */
if  ( myLBTS < true LBTS )
    { myLBTS = true LBTS; }
```

Table 1. End-host Operational Semantics.

```
/*  Recvd startLBTSmsg(i)
    from a child or parent:

    Note: if i < myColor then no
    action taken.  This is how
    LBTS computations are merged.
*/

startColor = myColor + 1;
while  ( startColor <= i )  {
    init SR(startColor) counters;
    init local LBTS(startColor);
    mcast startLBTSmsg(startColor) to
      all other parent or child links;
    myColor = startColor;
    startColor++;
}
```

Table 2. Router Start LBTS Operational Semantics.

```
/*  Recvd myLBTSmsg(i) from child:  */

Compute local min LBTS(i);
Sum SR(i) counters;
if  (all children reported LBTS(i))  {
    if  (I'm not the root)  {
        send myLBTSmsg(i) to parent;
    }  else  {
        /*  I'm the root  */
        if  (SR(i) cntrs sum to zero) {
            mcast trueLBTSmsg(i)
              to children;
        }
        /*  else wait for corrLBTSmsgs
            to fix things
        */
    }
}
```

Table 3. Router Compute LBTS Operational Semantics.

```
/* Recvd corrLBTSmsg(i) from child: */

Compute local min LBTS(i);
if  ( I'm not the root )  {
    /*  Don't bother to increment my
    Recv counter since only the
    root's Recv counter matters.
    */
    send corrLBTSmsg(i) to parent;
    /* Could rate-limit to prevent
       root bottleneck.
    */
}  else  {
    /*  I'm the root  */
    increment Recv counter;
    if  (SR(i) cntrs now sum to zero) {
        mcast trueLBTSmsg(i)
          to children;
    }
    /*  else wait for more corrLBTSmsgs
        to fix things
    */
}
```

Table 4. Router LBTS Correction Operational Semantics.

[10] F. Mattern. Efficient distributed snapshots and global virtual time algorithms for non-FIFO systems. *J. Parallel and Distributed Computing*, 18(4):423–434, August 1993.

[11] T. D. Modeling and S. Office. The High Level Architecture. *http://hla.dmso.mil/*, 2000.

[12] C. Pham and R. Bagrodia. Building parallel time-constrained HLA federates: A case study with the PARSEC parallel simulation language. 1997.

[13] J. Steinman, C. Lee, L. Wilson, and D. Nichol. Global virtual time and distributed synchronization. In *IEEE 9th Workshop on Parallel and Distributed Simulation*, pages 139–148, June 14–16 1995. Lake Placid, NY.

[14] D. Tennenhouse, J. Smith, W. Sincoskie, D. Wetherall, and G. Minden. A survey of active network research. *IEEE Communications Magazine*, 35(1):80–86, January 1997.

[15] S. Zabele and T. Stanzione. Interest management using an active network approach. In *Spring Simulation Interoperability Workshop*, March 26-31 2000.