Joining a Real-Time Simulation: Parallel Finite-State Machines and Hierarchical Action Level Methods for Mitigating Lag Time

Jianping Shi
Norman I. Badler
Michael B. Greenwald
Department of Computer & Information Science
University of Pennsylvania
Philadelphia, PA 19104-6389
215-898-1976
jshi@graphics.cis.upenn.edu
badler@central.cis.upenn.edu
greenwald@cis.upenn.edu

Keywords:
DVE; simulation; pilot/drone; checkpoint/restart; action level of detail; dead reckoning; avatar

ABSTRACT: *Distributed virtual environments, which simulate an actual physical or imaginary world on a network and allow multiple participants to interact simultaneously with one another within it, are becoming increasingly important for both research and practical purposes. As the number of participants and the amount of information exchanged among participants increase, it is crucial to large-scale distributed virtual environments to overcome bandwidth limitations and resolve network latency and synchronization problems. We present a new framework, called MELD, for modeling distributed virtual environments using the pilot/drone paradigm, which allows each host to locally model remote entities in order to resolve latency problems and improve responsiveness. Our approach uses shared event queues and a cache coherence protocol to synchronize the pilot/drone. To further improve the system's scalability, interest management is used to filter unneeded data before a host receives it for processing. The partition, however, introduces the problem of dynamically joining a group in a real-time simulation. We address this problem by presenting a checkpoint/restart mechanism based on an action hierarchy and a parallel finite-state machine structure. Additionally, ALOD (action level of detail) is employed to mitigate the lag between pilot and drone at any joining time.*

1. Introduction

A virtual environment (VE) is an interactive computer model that simulates an actual physical or imaginary world. Users can navigate within this virtual world, experience a logical series of events, and interact with 3D objects simulated by the system as well as with other users. A simulation model can be specified by a set of *states* and *events*. The execution of a simulation includes advancing the simulated virtual time, mimicking the occurrence of the events happening up to and including the simulation time, and calculating the effects of these events to modify the states and schedule newly generated events [1]. We are interested in the actions of human participants in a VE — entities that have significant non-linear movements and action vocabularies.

A distributed system is a collection of sequential processes P_1, P_2, ..., and P_n, and a network of unidirectional communication channels. Each channel connects a pair of processes and delivers messages between them [2]. Distributed simulations typically provide more overall host processing power and more storage space than sequential simulation systems, and thus support more simulation entities and more detailed appearance and behaviors of entities within the system. The distributed structure also allows multiple users located at geographically different sites to perceive the illusion of a single, coherent virtual world, to interact closely with each other as well as with the environment, and to consistently share the same experience.

Distributed virtual environments (DVE) are seeing increased use for a wide range of applications such as military simulations, education and training systems, virtual teleconferencing, collaborative modeling and engineering, and multi-user networked video games [3, 4, 5, 6, 7, 8]. The trend of faster processors, more powerful computer graphics hardware and software packages, and higher-capacity networks makes it possible for large-scale distributed virtual environment to contain well over 100,000 dynamic entities [9].

One of the critical issues facing DVE is scalability. As the number of participants and the amount of information exchanged among participants increase, it is crucial to large-scale distributed virtual environments to overcome bandwidth limitations and resolve network latency and synchronization problems. Many mechanisms have been developed in the literature to improve the bandwidth requirements, end-to-end latency, and scalability of DVE systems. Dead reckoning [20] is a promising technique that reduces the number of messages needed on a per-object basis. Interest management [5] is a technique that is critically needed to reduce the number of objects that each process observes.

However, as described in the next section, dead reckoning suffers from some intrinsic limitations. In particular, it is only applicable to a limited class of objects and behaviors, and it may not always achieve a significant reduction in the number of messages.

In this paper, we address these problems by generalizing the notion of dead reckoning to include much richer emulation of remote entities. However, this approach is complicated by interactions with interest management. We present a new distributed virtual environment framework, called *MELD* (Mitigating Entry Lag Delays), for modeling remote entities, synchronizing modeling processes at different hosts, solving the dynamic joining problem, and mitigating the lag time due to message delay. MELD adopts the pilot/drone paradigm and is thus efficient in terms of network bandwidth, latency, and responsiveness. It extends dead reckoning by specifying per-object extrapolation procedures.

The remainder of this paper is organized as follows. The next section provides a brief description of dead reckoning and interest management. In Section 3, we present our new framework MELD. Section 4 provides an example system that is used to compare our framework with dead reckoning. Finally, Section 5 summarizes the contributions of this work.

2. Dead Reckoning and Interest Management

Dead reckoning is used in SIMNET [3], DIS [4], and other distributed virtual environments that incorporate the DIS application protocol (such as NPSNET [5]). Dead reckoning is a technique to reduce network traffic. In a dead reckoning system, the database containing the initial state of the virtual world is replicated at all participating clients at the beginning of a simulation session. Then each client is responsible for maintaining its own replica of the database. The state of a remote entity is modeled by extrapolating from the last reported state sent from the entity's local host [20]. The entity's local host generates a new state update message and sends it to the remote hosts only if the discrepancies in the remote extrapolations exceed an error threshold.

SIMNET provides a concrete example. The virtual world in SIMNET consists of a collection of entities that interact with each other via events [3]. There is no central server process to schedule events or resolve conflicts between entity states among simulation nodes. A pilot is the graphical version of the avatar controlled on the user's own client; the drones are the avatar copies executing on other clients [10]. Note that sometimes *player* and *ghost* are used in place of pilot and drone to refer to exactly the same paradigm [5]. Each node is responsible for sending out the events caused by its pilot entities to other nodes, receiving event reports from other nodes, and calculating the effects of the received events on its own entities. Each simulation node is completely autonomous, and the simulation node sending out an event is not responsible for keeping track of the effects of the event on other nodes.

Between receipt of state update messages a local node extrapolates from the last reported states of remote entities that are of interest to the entities it is simulating, and uses the result of the extrapolation to generate displays for human crews or detection probabilities for automated crews [3]. In order to allow remote extrapolation, each simulation node must maintain a dead reckoning model that exactly corresponds to those used by remote nodes. The entity’s local node tracks both the actual states of their entities and the predicted states calculated with dead reckoning, and generate new state update messages before the discrepancies among the nodes become unacceptably large. Since it is not necessary to transmit state update messages at every frame in a dead reckoning system, bandwidth requirements can be reduced.

Dead reckoning merely reduces the per-entity bandwidth, but as the number of entities increases, the bandwidth still scales linearly. In a large-scale distributed virtual environment, interest management is critical to improve the system's scalability by filtering unneeded data before a client receives it for processing. For example, Macedonia et al [5] show that without interest management, NPSNET-IV can accommodate a maximum of 300 players on an Ethernet LAN (10 Mbps, saturated at roughly 70% utilization) without modification to the DIS protocol. However, in an exercise containing 1000 or more active entities, the number of entities that each player is interested in (those that are in the neighboring geographical regions and those that belong to related organizations) may be well below 300. If each player only receives and processes data packets from entities that it cares about, it is then possible to support many more players with the same underlying hardware capacity.

Reducing the traffic requirement by dead reckoning has two limitations: First, in order to support large-scale distributed simulations, dead reckoning protocols have to accept imperfect remote modeling and potential discrepancies. Second, dead reckoning uses simple extrapolation schemes to predict future location of remote entities, so it is effective only in modeling simple, predictable, and continuous motions or actions, such as the change of position of a tank, and does not apply to modeling unpredictable, complex, discrete actions, such as rich human behaviors.

3. MELD

3.1 Overview of MELD

MELD uses a pilot/drone paradigm similar to the pilot/drone paradigm used in SIMNET, but rather than using dead reckoning to model drone-behavior, we fully emulate the entity on *every* node. This is a reasonable trade-off considering the rapid increase in processing power relative to commodity network speeds. Given identical code on every node, we synchronize by simply ensuring that the set of external inputs seen by the pilot are also seen by each drone in the same order. Each avatar maintains a shared input event queue which remains consistent by using standard cache-coherency protocols.

Processor resources are cheap but not free. Network traffic is reduced, but not eliminated. There is still a need for interest management. Unfortunately, joining an interest group now means migrating a copy of the entire pilot process to the new drone. Fortunately, MELD can exploit its representation of pilots and drones to avoid much of the problems and complexity of process migration. However, migration still may take a non-negligible time, and after joining the interest group a drone may lag behind the pilot by a constant time. We propose a technique, Action Level of Detail (ALOD), to allow us to narrow the lag by accelerating the drone.

3.2 Components of MELD

MELD uses PaT-Nets (Parallel Transition Networks) to specify simulations in our framework. PaT-Nets are essentially finite automata executing in parallel. In our system, PaT-Nets execute in the *Jack* environment [11, 12]. Together with the *Jack* API, they provide an intuitive interface to control simulation and behavior of processes and agents in *Jack*.

The *node* is the basic building block of the PaT-Net [12]. There are several different types of nodes, but each has a similar structure and behavior. Each node has an associated *action*, and the *transitions* between nodes determine the path through the PaT-Net. Transitions can be randomly assigned, weighted with probability, or given as a set of ordered *conditions* from which the first valid condition will be selected. Conditions and actions can manipulate a set of *local variables*. A set of *monitors* adds control within the PaT-Net.

We then define a *logical process* (LP) as follows:

- An LP is a collection of active PaT-Nets, which are advanced at a fixed frame rate.

- Each LP is used to control an avatar's behavior, whether it is a pilot or a drone.

- An LP always has a distinguished PaT-Net, which we call the *behavior manager*. We will explain its responsibilities later on.

3.3 Pilot/drone synchronization

Figure 1 illustrates the pilot/drone paradigm. For simplicity, we assume avatar A_i is controlled by LP_i under the instructing of $User_i$ sitting in front of $Host_i$ (in a real application a single user may command multiple avatars via multiple LPs), and each host is observing the activities of all avatars (we will see later that users can choose to observe only a subset of avatars that they are interested in). In the figure, shaded circles represent pilot LP, blank circles represent drone LP, and dotted arcs represent the synchronization between pilot LP and its corresponding drone LPs.

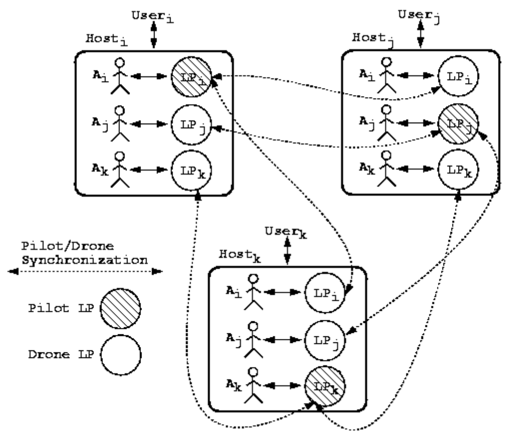

Figure 1: The pilot/drone paradigm

We associate a shared *event queue* with each avatar in the virtual world to synchronize its pilot and drone LPs. Each event queue has a fixed owner, which is defined as the pilot LP of the associated avatar. To implement the shared event queue structure, we use a fixed-owner, directory-based invalidate protocol similar to that used in many hardware or software based DSM (distributed shared memory) systems [13, 14, 15]. Directory-based coherency reduces network traffic because it does not use a broadcast scheme for one LP to send invalidate/update messages to all other LPs, which usually generates network traffic that is proportional to the number of LPs squared (M^2). Invalidate coherency protocols [21] require a writer (here pilot LP) to acquire exclusive access to a shared variable before writing. This is accomplished by invalidating all cached copies of the variable. In contrast, update or broadcast coherency protocols [21] make sure that all cached copies are consistently updated after each write. Invalidate protocols reduce bandwidth requirements, but have the potential to increase latency until up-to-date values are available at all nodes. “Fixed-owner” refers to the ownership of the directory entry for each shared variable, and does not imply any access privileges. Both the owner and non-owner LPs can write and read a state variable. However, when a state variable is flushed from a node’s cache, the node can later find the current writer (with the latest value) by querying the owner’s node.

One should note that invalidate protocols are typically a lazy, consumer-initiated (pull) communication [16]. Update protocols, on the other hand, send the new values to all remote copies that are consequently updated, whenever an LP writes a shared variable. Obviously the latter mechanism imposes an eager, producer-initiated (push) communication. Our framework allows the programmer to choose lazy or eager update for each individual shared state variable, in order to trade off bandwidth for latency according to their specific requirements.

3.4 Joining an interest group

When a drone joins a pilot’s interest group, we must capture the dynamic state of the pilot’s logical process. Capturing the state of a logical process, a problem equivalent to process migration, has been attracting a lot of research effort. Two promising approaches to capture dynamic state are checkpoint/restart mechanisms (e.g. process introspection [17]) and logging mechanisms (e.g. latecomer accommodation service [18]). Both approaches require programmer effort to modify the code, and are often platform dependent.

In this section we exploit PaT-Net semantics to present an efficient, yet easy-to-use, mechanism to capture the dynamic state of a logical process. Our approach uses the checkpoint/restart mechanism, but it requires little programmer effort, is platform-independent, and does not introduce any extra run-time overhead to the original PaT-Nets. In order to make it easier and less error-prone to checkpoint the dynamic state of an LP, we organize the LP into a hierarchy of active PaT-Nets. We will introduce two rooted tree structures: static behavior trees and dynamic behavior trees.

The static behavior tree is used to statically describe an avatar's comprehensive behavior. By “comprehensive” we mean that all possible actions of an avatar are included in the tree. The root of the static behavior tree for avatar A_i is A_i's behavior manager, which is responsible for reading the shared event queue associated with A_i and taking appropriate actions (e.g. instantiating a new PaT-Net) according to the events contained in the event queue. All the actions that A_i is capable of doing (called A_i's “capabilities” in [19]) are inserted into the tree as the root's children. Some actions are very complex and have other complex or primitive actions as their subactions. We add another level of action nodes in our static behavior tree to depict the subaction relationship among A_i's capabilities. Therefore the height of A_i's static behavior tree is at most two, and all the nodes at level 1 form the comprehensive set of actions that A_i can perform.

The root of the dynamic behavior tree for avatar A_i is also A_i's behavior manager, but the tree is used to keep track of A_i's dynamic behavior, i.e., the actions A_i is currently performing. From the PaT-Net point of view, the tree includes all the active PaT-Nets controlling A_i's actions. In the *Jack* system there is an *active PaT-Net list* that manages the advancement of all active PaT-Nets. Therefore a one-to-one mapping exists between the nodes of dynamic behavior trees and the PaT-Nets in the active PaT-Net list.

Our checkpoint/restart mechanism is based on the PaT-Nets hierarchy. The entire procedure is divided into two phases: checkpointing an LP at its owner host, and restarting it at a remote host.

Upon receiving a *join* request from $Host_j$, avatar A_i performs the following steps to checkpoint its dynamic activities:

(1) Check if $Host_j$ is already in the interest group of A_i. If yes, return.

(2) Externalize A_i's physical information (e.g. global position, or joint angles) to a stream and send it to $Host_j$.

(3) Externalize A_i's dynamic behavior tree to a stream and send it to $Host_j$.

(4) For each active PaT-Net in A_i's dynamic behavior tree, externalize it to a stream and send it to $Host_j$.

(5) Set flags so that $Host_j$ will be notified of every update to A_i's shared event queue, and the drone LP (being reconstructed using the procedure discussed later on) running at $Host_j$ will be synchronized with A_i's pilot LP via the shared event queue.

After receiving all of the packets generated by these steps, $Host_j$ can restart a replica (i.e., a drone LP) of A_i's logical process and synchronize its execution with that of the pilot LP. Restarting an LP is functionally the reverse of checkpointing it, however, care must be taken to handle packets that arrive out of order. In order to process the incoming packets in the correct order, we divide the life cycle of a drone LP into five different stages, as shown in Figure 2.

At each stage, only certain types of packets are allowed to be processed. For example, at the joining stage, only the physical information packet can be processed. If the dynamic behavior tree packet or a PaT-Net internal state packet arrives at this stage, it will be buffered for later processing; only at the physical information restored stage can the dynamic behavior tree packet be processed. If a PaT-Net internal state packet is received while the receiving LP is in this state, the packet will also be buffered; and all the PaT-Net internal state packets can only be processed after the dynamic behavior tree has been reconstructed.

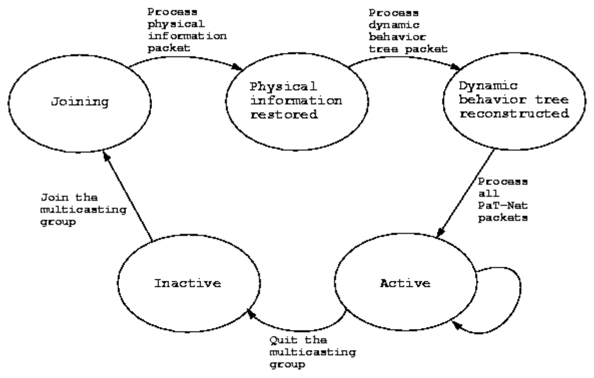
Process physical information packet
Process dynamic behavior tree packet
Joining
Physical information restored
Dynamic behavior tree reconstructed
Join the multicasting group
Process all PaT-Net packets
Inactive
Active
Quit the multicasting group

Figure 2: The life cycle of a drone LP

After all the PaT-Nets in the dynamic behavior tree are successfully internalized, they will be added to the active PaT-Net list in order to get advanced at the subsequent simulation ticks, and the stage of the drone LP will be changed to *Active*, which marks the completion of our restarting procedure.

3.5 Lag mitigation

When a drone LP is first restarted, there is a lag between its state and that of the pilot LP due to message delay. This delay consists of the time it takes to generate, deliver, and process messages containing checkpoint streams between two logical processes. Figure 3 illustrates this lag, and a general way of mitigating it. For simplicity, we assume that the local clocks of all logical processes are *perfectly synchronized*, i.e., for all *t*, and any two processes *p* and *q*, $C_p(t) = C_q(t)$, where $C_p(t)$ and $C_q(t)$ are the readings of the local clocks of processes *p* and *q*, respectively, at real-time *t* (this constraint will be relaxed later on). In Figure 3, the pilot LP checkpoints its dynamic state at time t_1, while the drone LP does not finish restoring it until t_2. Since the pilot LP proceeds during the time period from t_1 to t_2, the drone LP will be behind the pilot LP at time t_2 when it is just restarted. To mitigate such lag between the drone and pilot LPs, we must somehow accelerate the execution of the drone LP from t_2 on, until both are synchronized (t_3 in the figure), and then resume normal execution for the drone LP. Therefore, we have two questions to answer:

(1) How to accelerate the execution of the drone LP so that it can catch up with the pilot LP?

(2) How does the drone LP know that it is in synchronization with the pilot LP? In other words, when to stop the acceleration session and resume the normal execution for the drone LP?

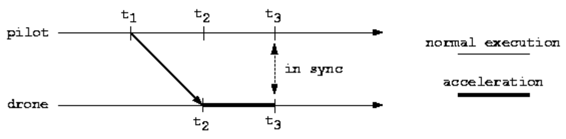

Figure 3: Lag mitigation

To solve the first problem, we introduce a new paradigm for distributed simulation: *action level of detail (ALOD)*. ALOD is analogous to geometric level of detail. A number of PaT-Nets at different levels of detail are provided to describe the same action for an avatar, and so is a mechanism to switch back and forth among different levels of detail on the fly. Assume during its normal execution a logical process uses the PaT-Nets from the highest level and with the greatest degree of detail for all the actions in its dynamic behavior tree. When a drone LP is just restarted, it will switch to a lower level that consumes fewer resources, shorter time, and smaller number of frames for each action due to its lower quality. Then after it has been synchronized with the pilot LP at the end of the acceleration session, the drone LP will switch back to the highest level of detail and resume its normal execution.

It may seem that we can simply utilize our checkpoint/restart mechanism to switch back and forth among different levels of detail. Unfortunately it is not that straightforward, because the source and target LPs here are not identical. That is, we checkpoint the dynamic state of one LP, but try to restore the state of another LP later on from this checkpoint. Therefore, to enable the usage of our checkpoint/restart mechanism in the ALOD transition procedure, we have to enforce the following constraints on the relationships among behavior trees and PaT-Nets at all possible levels of detail: for all avatars A_k, and all $i \neq j$,

- $ALOD^i$ (the ith level of detail) and $ALOD^j$ have exactly the same structure for their static behavior trees. That is, A_k has the same capabilities at both $ALOD^i$ and $ALOD^j$, and the subaction relationships among the actions in both static behavior trees at $ALOD^i$ and $ALOD^j$ are also the same. Therefore, at any time a transition is made from $ALOD^i$ to $ALOD^j$, it is guaranteed that the dynamic behavior trees at both levels of detail have exactly the same structure too.

- Suppose PN^i and PN^j are two PaT-Nets used to simulate the same action of avatar A_k at $ALOD^i$ and $ALOD^j$, respectively. Then there must be bi-directional mappings between the set of local variables of PN^i and that of PN^j, and between the set of nodes of PN^i and that of PN^j. However, their internal transitions, actions and conditions can be defined differently in order to achieve different levels of detail, but precaution must be taken to ensure that the mapping between PN^i and PN^j is well defined so that the dynamic state of PN^j can be inferred from that of PN^i in a straightforward way.

To solve the second problem, we need to know the lag between the drone LP and the pilot LP. The drone LP can exploit this information to decide which lower level of detail it will switch to, and how long it will stay at this lower level of detail before switching back to the normal level (i.e., the highest level). Assume local clocks of all logical processes are perfectly synchronized. The drone can compute the lag if, in Figure 3, the pilot timestamps its messages containing checkpoint streams, and also sends an extra value τ along with the checkpoint information to the drone, where τ is an estimate of the interval of time between two consecutive simulation ticks at the pilot. Then, we have

$$d = t_2 - t_1,$$
$$h = \frac{d}{t},$$

where δ is the message delay, and η is the number of frames by which the drone is behind the pilot at time t_2.

Based on the value of η, the drone can decide whether or not it will switch to a lower ALOD, and if yes, which ALOD it will switch to in order to catch up with the pilot. If η is very small, say only a couple of frames, the drone may not bother to switch ALODs because the transition itself would introduce a certain amount of overhead. Otherwise the drone will select an ALOD to switch to according to the value of η. Then finally, it is necessary to calculate the length of the acceleration session. Suppose the normal execution is at $ALOD^i$, and the drone decides to go to $ALOD^j$ to accelerate its execution, then we have

$$r = \frac{\text{the simulation time at } ALOD^i}{\text{the simulation time at } ALOD^j},$$

where ρ is the speedup of $ALOD^j$ over $ALOD^i$. Hence, the length of the acceleration session is

$$h' = \frac{(h + 2 \cdot l) \cdot r}{r - 1},$$

where η' is the number of frames contained in the acceleration session, and λ is the overhead introduced by the transition procedure from one ALOD to another. Therefore, the complete lag mitigation procedure works just as follows: the drone switches to $ALOD^j$ at time t_2, stays at $ALOD^j$ for η' number of frames, and then switches back to $ALOD^i$ to resume its normal execution.

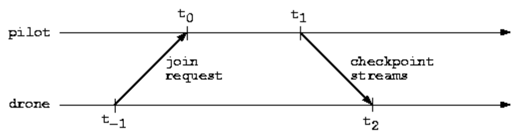

Figure 4: Measuring the message delay when local clocks are not perfectly synchronized

Now we relax the constraint about the local clocks by only assuming that our distributed simulation system is *synchronous* [22]. Under this relaxed constraint, the message delay δ is not simply equal to $t_2 - t_1$ any more because the local clocks of the pilot and the drone are not perfectly synchronized. However, we can get rid of the clock offset by timestamping the *join* request sent from the drone to the pilot too, as shown in Figure 4. Hence,

$$d = \frac{(t_0 - t_{-1}) + (t_2 - t_1)}{2}.$$

Note that t_{-1} and t_2 are the readings of the drone's local clock, and t_0 and t_1 are the readings of the pilot's local clock.

4. Example System

Consider an avatar's behavior in a shoot-out video game. There are four states associated with the avatar: standing, shooting, dodging, and falling down. Initially, the avatar is standing there with a pistol in his holster. When the player of the avatar instructs him to shoot, the system will change the avatar to the shooting state, at which point the avatar will perform a sequence of actions including drawing the pistol, aiming it at the target, and firing. In either of those two states, if the avatar sees somebody shooting at him, he will immediately transition to the dodging state, trying to protect himself from the bullet first (obviously he is not a brave cowboy). At any of those three states, if a bullet's trajectory intersects any polygon composing the avatar's body (e.g. the avatar did not see the bullet coming, or he could not get out of the way in time), the system will move to the falling down state, and the avatar will fall down to the ground (he is not a strong one either).

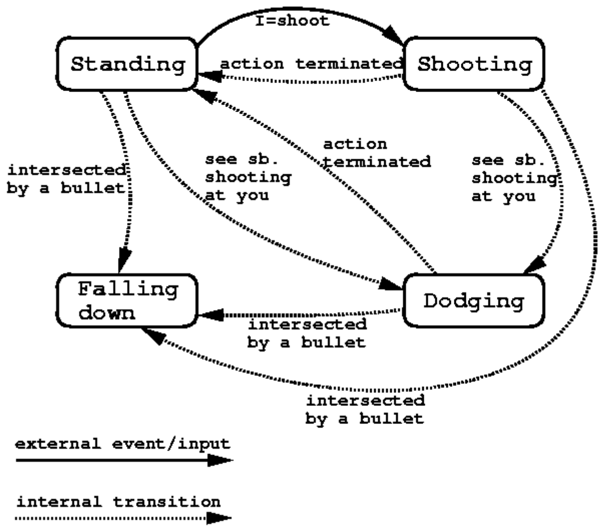

Figure 5: An example of a pilot LP, in both dead reckoning and MELD. Note that the corresponding drone LP in MELD is also identical to this

Figure 5 displays a finite-state machine representing this simple model. The external events/inputs are shown as solid arcs of the finite-state machine, and the internal transitions are shown as dotted arcs in the figure. This example has only one external event, which is that the user inputs the shooting instruction; all the others are internal transitions that take place when certain conditions on the state set of the model are satisfied. For example, at the standing state, if the intersection of a bullet's trajectory and the avatar's body is detected, the system will switch to the falling down state. Note that "intersected by a bullet" and "see somebody shooting at you" are internal because they are the result of outputs of other drone LPs or local pilots. This graph is a collection of related behaviors, not just an animation mechanism (e.g. posture graph [23]).

Note that in our framework both pilot and drone LPs are identical because we duplicate both the static database and the modeling processes at all of the clients. In contrast, dead reckoning systems model the drones by using simple extrapolation procedures to predict the future state of an entity. Figure 6 displays the finite-state machine representing the corresponding drone LP of that shown in Figure 5. We can see that all of the transitions are now external events, because the change of the state (discontinuity in the state space) cannot be predicted using simple extrapolation schemes.

In MELD, an event message is needed to be sent to the drone to induce the state transition only when the player inputs a shooting instruction. In the absence of such input, no network communication is necessary, and no discrepancies between pilot and drone exist. On the other hand, in a dead reckoning system, every time there is a state change, a state update message will be sent to the drone to induce the state transition. The situation gets even worse if we consider how poorly the simple extrapolation schemes can predict within some states. For example, pure position dead reckoning is obviously not effective in modeling the complex shooting action, which includes a sequence of subactions such as drawing the pistol, aiming, and pulling the trigger.

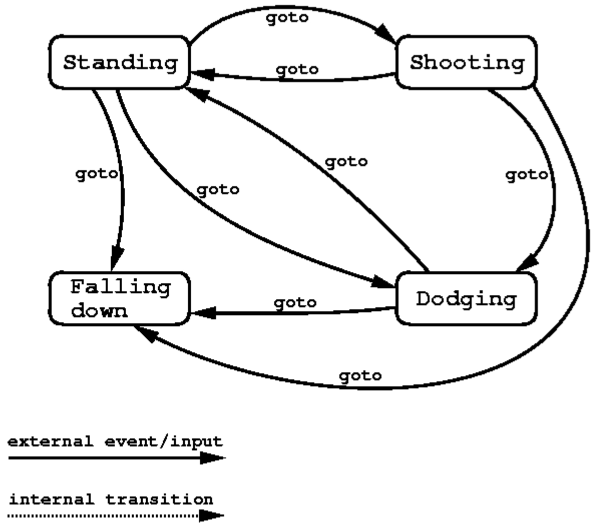

Figure 6: An example of a drone LP in dead reckoning

This example illustrates several advantages our framework has over dead reckoning. First, in the absence of external events, discrepancies between pilot and drone do not exist because we use per-class custom extrapolation procedures (which can be simple copies of the pilot's code) to model remote entities. Second, network bandwidth requirements are reduced compared to a dead reckoning system because, in our framework, state update messages are sent only on external events/inputs. Finally, MELD can model discontinuous and unpredictable actions such as rich human behaviors because, in our framework, the drones manipulate the entire state (including internal state), not just the external behaviors.

5. Conclusions

The contributions of our work can be summarized as follows:

- Provides a new framework to implement distributed virtual environments, which adopts the pilot/drone paradigm and is thus efficient in terms of network bandwidth, latency, and responsiveness.

- Extends dead reckoning by specifying per-object extrapolation procedures. The extrapolation procedure in the drone is identical to the simulation specification in the pilot. MELD drones model the entire state of each object of interest, not simply the subset of states that represent external behavior. Therefore MELD pilots need only send updates when external events change the state of the pilot (reducing communication needs in the common case). In the absence of external events, discrepancies between pilot and drone do not exist. MELD can therefore model richer behaviors such as discontinuous and unpredictable motions or human actions.

- Incorporates interest management to filter unneeded data to further improve the system's scalability, and addresses the dynamic joining problem introduced by interest management using a checkpoint/restart mechanism, which is based on the action hierarchy and the parallel finite-state machine structure.

- Provides a straightforward yet efficient mechanism for an agent to smoothly join an ongoing simulation. This mechanism includes an extension of geometric level of detail to action level of detail (ALOD).

- Extends the notion of uniprocessor time-critical human animation [24]. The hierarchy of PaT-Nets we described provides a convenient framework for the multiple motion generators required by each human model.

6. Acknowledgments

This research is partially supported by Office of Naval Research K-5-55043/3916-1552793, NSF IIS99-00297, and NASA NRA NAG 5-3990.

7. References

[1] Alois Ferscha and Satish K. Tripathi: "Parallel and Distributed Simulation of Discrete Event Systems" Technical Report CS-TR-3336, University of Maryland, August 1994.

[2] Ozalp Babaoglu and Keith Marzullo: "Consistent Global States of Distributed Systems: Fundamental Concepts and Mechanisms" In Sape Mullender, editor, Distributed Systems. ACM Press, 1993.

[3] Duncan C. Miller and Jack A. Thorpe: "SIMNET: The Advent of Simulator Networking" Proceedings of the IEEE, 83(8), August 1995.

[4] J. Mark Pullen and David C. Wood: "Networking Technology and DIS" Proceedings of the IEEE, 83(8), August 1995.

[5] M. Macedonia, M. Zyda, D. Pratt, P. Barham and S. Zeswitz: "NPSNET: A Network Software Architecture for Large-Scale Virtual Environments" Presence, 3(4): 265-287, Fall 1994.

[6] C. Carlsson and O. Hagsand: "DIVE — A Platform for Multiuser Virtual Environments" Computers and Graphics, 17(6): 663-669, 1993.

[7] Gurminder Singh, Luis Serra, Willie Png and Hern Ng: "BrickNet: A Software Toolkit for Network-Based Virtual Worlds" Presence: Teleoperators and Virtual Environments, 3(1): 19-34, Winter 1994.

[8] Chris Greenhalgh and Steve Benford: "MASSIVE: a Distributed Virtual Reality System Incorporating Spatial Trading" Proceedings of the 15th International Conference on Distributed Computing Systems, pages 27-34, May 30-June 2 1995.

[9] Sandeep Kishan Singhal: "Effective Remote Modeling in Large-Scale Distributed Simulation and Visualization Environments" PhD thesis, Department of Computer Science, Stanford University, 1996.

[10] Living Worlds, 1997. http://www.vrml.org/WorkingGroups/living-worlds/draft_2/index.htm.

[11] N. Badler, C. Phillips and B. Webber: "Simulating Humans: Computer Graphics Animation and Control" Oxford University Press, New York, 1993.

[12] John P. Granieri, Welton Becket, Barry D. Reich, Jonathan Crabtree and Norman I. Badler: "Behavioral Control for Real-Time Simulated Human Agents"

Symposium on Interactive 3D Graphics, ACM Press, April 9-12, 1995.

[13] Agarwal, R. Bianchini, D. Chaiken, K. Johnson, D. Kranz, J. Kubiatowicz, B. Lim, K. Mackenzie and D. Yeung: "The MIT Alewife Machine: Architecture and Performance" Proceedings of the 22nd Annual International Symposium on Computer Architecture, pages 2--13, June 1995.

[14] D. Lenoski, J. Laudon, K. Gharachorloo, W. Weber, A. Gupta, J. Hennessy, M. Horowitz and M. Lam: "The Stanford Dash Multiprocessor" IEEE Computer, pages 63--79, March 1992.

[15] Kirk Johnson, M. Kaashoek and D. Wallach: "CRL: High-Performance All-Software Distributed Shared Memory" Proceedings of the 15th Symposium on Operating Systems Principles, December 1995.

[16] Daniel E. Lenoski and Wolf-Dietrich Weber: "Scalable Shared-Memory Multiprocessing" Morgan Kaufmann Publishers, 1995.

[17] Adam J. Ferrari, Stephen J. Chapin and Andrew S. Grimshaw: "Process Introspection: A heterogeneous checkpoint/restart mechanism based on automatic code modification" Technical Report CS-97-05, Department of Computer Science, University of Virginia, March 1997.

[18] Goopeel Chung, Prasun Dewan and Sadagopan Rajaram: "Generic and Composable Latecomer Accommodation Service for Centralized Shared Systems" 1998 IFIP Working Conference on Engineering for Human-Computer Interaction, September 1998.

[19] N. Badler, R. Bindiganavale, J. Allbeck, W. Schuler, L. Zhao, and M. Palmer: "Parameterized Action Representation for virtual human agents" In J. Cassell (ed.), Embodied Conversational Agents, MIT Press, 2000.

[20] Sandeep Singhal and Michael Zyda: "Networked Virtual Environments: Design and Implementation" Addison-Wesley Pub Co, 1999.

[21] John Hennessy and David Patterson: "Computer Architecture: A Quantitative Approach, Second Edition" Morgan Kaufmann Publishers, 1996.

[22] Vassos Hadzilacos and Sam Toueg: "Fault-Tolerant Broadcasts and Related Problems" In Sape Mullender, editor, Distributed Systems. ACM Press, 1993.

[23] J. P. Granieri, J. Crabtree, and N. I. Badler: "Production and Playback of Human Figure Motion for Visual Simulation" ACM Transactions on Modeling and Computer Simulation, 5(3): 222-241, July 1995.

[24] J. P. Granieri: "Time-Critical Human Figure Animation for Interactive 3D Visual Simulation Applications" PhD thesis, Department of Computer & Information Science, University of Pennsylvania, 2000.

Author Biographies

JIANPING SHI is a PhD candidate in computer and information science at the University of Pennsylvania. His research interests include distributed virtual environments, computer networking and computer animation. Shi received a BE in computer science from the University of Science and Technology of China in 1993; and an MS in computer science from the University of Pennsylvania in 1996.

NORMAN I. BADLER is a Professor of Computer and Information Science at the University of Pennsylvania and has been on that faculty since 1974. Active in computer graphics since 1968 with more than 180 technical papers, his research focuses on human figure simulation and animation control. Badler received the BA degree in Creative Studies Mathematics from the University of California at Santa Barbara in 1970, the MSc in Mathematics in 1971, and the Ph.D. in Computer Science in 1975, both from the University of Toronto. He is Co-Editor of the Journal *Graphical Models and Image Processing* and also directs the Center for Human Modeling and Simulation at UPenn.

MICHAEL B. GREENWALD is an Assistant Professor in the department of Computer and Information Science, at the University of Pennsylvania. His research interests include synchronization in distributed systems, congestion control in large-scale computer networks, and performance measurement and analysis of computer systems. Greenwald received an S.B. in Mathematics from the Massachusetts Institute of Technology and a PhD in Computer Science from Stanford University.