By Jesse Aronson
Gamasutra
September 19, 1997

Features

Dead Reckoning: Latency Hiding for Networked Games

Programmers who attempt to develop games for network play quickly discover the problems of dealing with network latency. Latency is, quite simply, the time it takes packets of information to get from one computer to another across the network and is a function of the path traveled by packets passing between the computers.

Latency is not a problem for playing slow moving, turn-based games such as chess or Go across a network. On the other hand, games that require high levels of fast-paced interaction between players, known as "twitch" games, can be severely degraded by typical WAN or even LAN latencies. This becomes problematic, as players expect the level of performance of distributed games to approximate that of single computer, single player games--and players want twitch games.

Current commercial solutions from Internet gaming vendors for the most part rely on dedicated network segments to provide end-to-end latencies in the 150 to 200 ms range. Dedicated networks are a viable solution to the problem, however it would be better if networked games operated in a fluid and responsive fashion in a standard Internet environment, where latencies are less predictable and generally larger.

Fortunately, your tax dollars have been at work developing technical solutions for networked games. The Dept. of Defense has invested heavily in distributed simulation for military training, most notably in the development of the Distributed Interactive Simulation (DIS) protocol, which developed out of the Defense Advanced Research Project Agency's (DARPA) SIMNET project.

DIS contains a technique for latency hiding and bandwidth reduction. That technique, called "dead reckoning," has been widely implemented and has been expanded in DARPA's Advanced Distributed Simulation architecture project into the notion of "predictive contracts." DIS originated as an environment for networking together tank simulators, and many DIS applications share common characteristics with twitch games.

The Dead Reckoning Concept

Dead reckoning is a form of replicated computing in that everyone participating in a game winds up simulating all the entities (typically vehicles) in the game, albeit at a coarse level of fidelity. The basic notion of dead reckoning is agreement in advance on a set of algorithms that can be used by all player nodes to extrapolate the behavior of entities in the game, and an agreement on how far reality should be allowed to get from these extrapolation algorithms before a correction is issued.

Under DIS, when a vehicle or entity is created, the computer that owns the entity sends out what is called an entity state protocol data unit (PDU) to all other computers on the network. The entity state PDU contains information that uniquely identifies the entity; information that describes the current kinematic state of the entity, including position, velocity, acceleration and orientation; and other information, such as the entity's damage level.

Last, the entity state PDU contains an identifier that tells all other nodes on the net which dead reckoning algorithm to use for this entity. When other computers participating in the distributed simulation receive this PDU, they create local copies of the specified type of entity. Thus, every node on the net begins to see this entity. After this, entity state PDUs are sent out at a minimum of one every five seconds.

Without some sort of extrapolation algorithm, the single entity state PDU sent at start-up would cause the entity to appear remotely, but the remote entity would be static; it would only move as additional PDUs describing its updated parameters were sent out. Thus, the motion of remote entities would seem choppy; they would stay still until the next PDU was received, and would jump to the location specified in the new PDU. This choppiness can be lessened by decreasing the time between PDUs, but this approach quickly exhausts available network bandwidth in scenarios with large numbers of entities.

With dead reckoning, however, after receipt of the first entity state PDU for an entity, each node on the net begins moving the entity by applying the agreed-upon dead reckoning algorithm. As long as the entity continues to move in a predictable fashion, it appears in a consistent, synchronized way on all nodes on the net with no further network traffic required.

Of course, simulation entities don't always move in a predictable fashion. The instant a player controlling an entity moves the control stick, the vehicle deviates from a smooth, algorithmically definable path. Under DIS, this is detected and handled by the computer that owns the entity.

The owner of an entity remembers the last time it put out an entity state PDU and also runs the dead reckoning algorithm based on that PDU. Thus, it has a copy of what all the other nodes on the network are seeing as well as the true, latest value.

The owning simulation compares the dead reckoning values to the true state of the entity as controlled by the player. If the dead reckoning and true state values differ by an amount that exceeds the agreed-upon dead reckoning threshold, a new entity state PDU is sent out to update the other nodes on the net. All nodes update their copies of the entity to reflect the new entity state PDU values, and dead reckoning begins again with the new data point.

Figure 1 shows an example of dead reckoning. At time t0, a simulation of an aircraft, shown on the left, first puts out a PDU informing all the computers on the network of the aircraft's existence and location. At this time, the position of the aircraft is synchronized at all computers on the network. From this point forward, all the computers on the network begin displaying the aircraft, and move it forward based on an agreed-upon algorithm. This is shown as a dashed line.

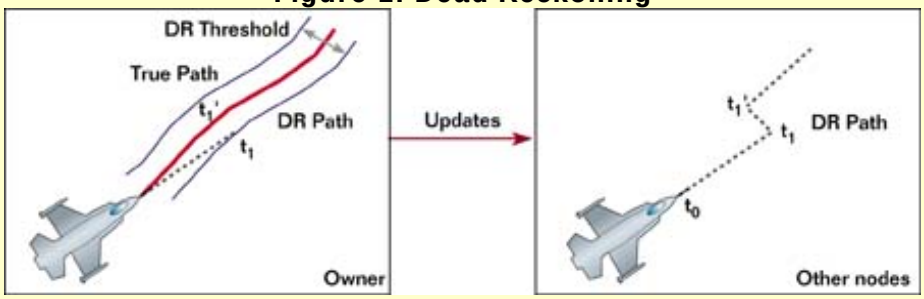
The owner of the aircraft, however, also moves the aircraft based on inputs from the user (for example, via a joystick). This is shown as the thick solid line on the left. The player controlling the aircraft sees motion along the thick solid line, while all other players on the net see motion along the dashed line. As long as the dead-reckoned position and the true position stay within a predefined threshold, shown via the thin black lines, no additional information is sent out on the network. Network bandwidth is conserved and remote computers show smooth motion based on the dead reckoning algorithm.

However, a small discrepancy between the owning simulation and all other simulations on the net exists. When the discrepancy gets bigger than the dead reckoning threshold, as at time t1, a new PDU is sent out. At this point, all computers on the network immediately resynchronize their copies of the aircraft to the new PDU data and reset their dead reckoning algorithms, as shown at t1'. Dead reckoning then begins again, as shown at t1' and beyond.

It is easy to see from the exaggerated view shown in Figure 1 that large dead reckoning thresholds can result in noticeable jerkiness of motion when new PDUs are received. On the other hand, small dead reckoning thresholds force more PDUs to be sent. The optimal values for dead reckoning thresholds are dependent on the type of application.

DIS simulations also typically use smoothing algorithms to lessen the apparent jerkiness as entities are updated from dead reckoned positions to new updated true positions. DIS simulations also typically apply dead reckoning to vehicle orientation as well as position. Orientation extrapolation uses a different set of algorithms from position extrapolation, although the two are based on identical concepts.

Figure 1. Dead Reckoning



Dead Reckoning Algorithms

Next, let's look at the actual dead reckoning algorithms DIS implements. There are nine standard algorithms, though two effectively shut off dead reckoning and several others duplicate each other, only using different coordinate systems. We'll examine the three most typical.

Figure 2 shows three DIS dead reckoning algorithms. These algorithms flow from basic physics. The first algorithm maintains an entity at the position specified in the entity's entity state PDU from t0. The second algorithm extrapolates the entity forward from its known t0 position based on its velocity at t0. The third algorithm also extrapolates forward from the entity's last known position, but uses both velocity and acceleration in the extrapolation.

The three algorithms shown in Figure 2 work well for extrapolating position. More sophisticated algorithms used in DIS also consider orientation (roll, pitch, and heading) of entities and even extrapolate moving parts of entities. For example, a tank that scans its turret back and forth while patrolling could make use of dead reckoning for both the tank's position and the angle of the turret on the tank.

Figure 2. DIS Dead Reckoning Algorithms

$$Position_{t1} = Position_{t0} \quad (\#1)$$

$$Position_{t1} = Position_{t0} + \vec{v}(t1 - t0) \quad (\#2)$$

$$Position_{t1} = Position_{t0} + \vec{v}(t1 - t0) + 1/2\vec{a}(t1 - t0) \quad (\#3)$$

Putting Dead Reckoning to Work

Listing 1 shows a fragment of a program that receives entity state PDUs from other programs on the network and displays all the entities it knows about, using dead reckoning algorithm 2 from Figure 2. The code is C++ style, though the class definitions, program initialization, and other details are not shown.

The program's primary loop first checks to see if there are any new packets available from the network (#1). DIS uses UDP for its communications, and so here we use a Java-like UDP socket class to access UDP packets. Next, the raw packets are parsed and

converted to entity state PDU objects, which contain information about remote entities' position, velocity, and identity.

If this entity is already known to the program (that is, its ID exists in `TankList`), the position and velocity for the entity are updated. Otherwise, a new entry in the TankList table is created (#3). The time of receipt is stored in the remote vehicle table (`ctime` is a function that returns the current time); this is required for dead reckoning.

After the new packet is received, the locations of all entities in `TankList` are updated using dead reckoning (#4) and are displayed on the screen (#5). Dead reckoning does not overwrite the received position, which may be needed for further dead reckoning at a later time.

Listing 2 shows a fragment of a program that sends information to the network about a tank it's simulating. It follows the rules of dead reckoning as well. The program repeatedly updates vehicle myTank based on inputs from the joystick controller. After each update, it checks to see if the dead reckoning threshold has been exceeded. If it has, it sends out a new PDU describing the updated state. The program sends one entity state PDU at startup to let other players on the network know that the vehicle `myTank` exists and that the program must remember the last state information and time it sent out.

Listing 1. Receiver Simulation with Dead Reckoning

```
main(){
  //Receiving Simulation
  DataGramSocket socket1;
  DataGramPacket packet1;
  EntityStatePDU espdu1;
  TankList remoteTanks;
  int i;

// Initialization code: open socket, etc.
// ... (Not shown here)

// Enter a loop, receiving and processing remote
// information forever
while(1){
//Receive a new packet
if (socket1.packetAvailable()){ #1
 socket1.receive(&packet1);
 espdu1.convertFromRawPacket(&packet1); //#2
 if (TankList.member(espdu1.entityID()) //#3
   TankList.update(espdu1,ctime());
 else
   TankList.addEntity(espdu1,ctime());
 }
...
```

Listing 2. Sending Simulation with Dead Reckoning

```
main(){
  //Sending Simulation
  Joystick stick1;
  DataGramSocket socket1;
  EntityStatePDU espdu1;
  tank myTank(initPosition);
  int i;


  // At init time, send a PDU and save info
  espdu1.initializeWithPosition(myTank.Position());
  socket1.send(espdu1.convertToRawPacket());
  lastStateSent = myTank;
  lastTimeSent = ctime();


  while(1){
    //Update my tank based on joystick
    myTank.calcNewPosition(JoyStick.read());


    //Calculate Dead Reckoned Position
    myTank.setDRposition(lastStateSent.position() +
       lastStateSent.velocity() * (ctime() - lastTimeSent);

    //Only send an update if DR threshold exceeded
    if (abs(myTank.position() - lastStateSent.position() > thresh){
      socket1.send(espdu1.convertToRawPacket());
      lastStateSent = myTank;
      lastTimeSent = ctime();
      }
    }
}
```

Extensions to Dead Reckoning

DIS uses dead reckoning primarily as a means to extrapolate position and orientation of vehicles, and it does so via polynomial algorithms. DIS applications have extended dead reckoning in one way to allow extrapolation of articulated parts of vehicles (for example, turrets on tanks).

The DARPA ADS architecture study recognized the value of dead reckoning and extended the concept to apply to all attributes of objects on the network and to a larger class of extrapolation. Extrapolations using these expanded definitions are referred to as predictive contracts.

For example, in a tank battle simulation, a predictive contract called "drive along road to waypoint" could be defined. Under this predictive contract, a computer simulating a tank would only have to send out one piece of information about a vehicle: it was at a certain position on a road, and it was going to drive down that road to a specified point.

As long as all other computers on the network agree on what it means to "drive down a road" (for example, that you drive on the right) and all the computers on the network know the definition of the road the tank is on, all the computers on the network can create consistent views of the tank without requiring any network traffic. Of course, if the

tank deviates from its planned path or changes state in any other unpredictable way, new state information for the tank would be sent out on the network. Additional predictive contracts could define when the tank turns its sensors on and off, when it sends out radio reports, and other attributes of the tank's behavior.

The advent of the Java model of distributed object computing allows a further refinement of predictive contracts. In the DIS model all dead reckoning algorithms are defined at compile time; there is no way for a simulation to create new dead reckoning algorithms at run time. With languages such as Java, however, a simulation could distribute updated predictive contracts across the network at will.

Make the Most of Dead Reckoning

Dead reckoning and predictive contracts offer dual benefits for networked simulations and games. They hide the latencies inherent in the network, and they offer the additional benefit of keeping traffic off the network by allowing simulations to transmit information only when really needed. In DIS, dead reckoning also allows entity state information be transmitted via best-effort communications (such as UDP/IP) rather than more expensive, reliable (such as TCP/IP) mechanisms. This is because dead reckoning can be used to smooth over gaps when packets are lost, albeit with some loss of synchronization across the network.

Dead reckoning is not free. It requires that every computer on the network run an algorithm to extrapolate each entity in the simulation scenario. Also, if all the entities in a simulation behave unpredictably all the time, dead reckoning offers little gain. However, it has been the experience in DIS that dead reckoning offers significant advantages in large scenarios with many computer-generated entities. In such situations, and where processor cycles can be traded off to reduce network use and apparent latency, dead reckoning can be a very effective optimization technique.

Jesse Aronson is a Principal Software Architect at Science Applications International Corp. He has been an active participant in defining the Dept. of Defense's next-generation architecture for modeling and simulation, and is currently technical director of a project to create a networked training simulation with hundreds of computers simulating thousands of vehicles. He can be reached at jaronson@std.saic.com.