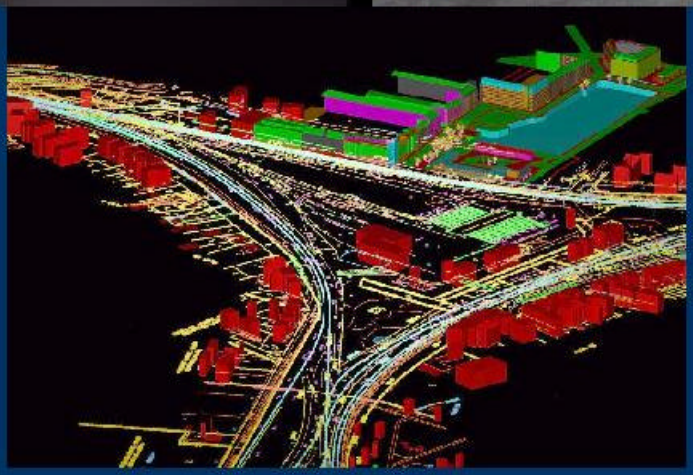
Synchronizing Distributed Virtual Worlds

Volume 3 in the *Simulation 2000* Series



Roger Smith

Title:	**Synchronizing Distributed Virtual Worlds**
Author:	**Roger D. Smith** Chief Scientist, ModelBenders LLC http://www.modelbenders.com/
Summary:	Simulations and virtual worlds are often designed to operate across local and wide area networks. This allows them to grow larger that a single computer can support. But more often it creates a distributed virtual world that can be experienced simultaneously by people around the world. When these distributed virtual worlds are created, it is necessary to install a mechanism that synchronizes the events that are executed in each computer. Many techniques for synchronizing distributed virtual worlds have been invented in research labs or engineered by people who needed an immediate solution. The methods fall into three general categories – independent, conservative, and optimistic. In this article we describe each of these techniques and methods for implementing them in software. Examples are used to illustrate how each technique works and what its limitations are. Volume 3 in the ***Simulation 2000*** series. **Intended Audience:** • Computer Game Programmers • Multi-user Web-site Developers • High School and College Students • Classroom Teachers **Outline of Material:** • Introduction • Discrete Event Simulation • Distributed Virtual Worlds • Independent Operations • Conservative Synchronization • Optimistic Synchronization • Recommended Applications • Excellent References

INTRODUCTION

Distributed virtual worlds are a specific implementation of simulation, graphics, and networking technologies that have been evolving for decades. It is only recently that each of these fields has become sufficiently advanced to support consumer products like networked computer games and multi-user web-sites. As these applications have grown in popularity, the developers have been searching for techniques to keep the distributed pictures of the virtual world synchronized. This is necessary to insure that each player or user is experiencing the same version of the world, with the same cause-and-effect relationships between observed phenomena.

Networked Multi-Player Gaming Example. Computer games are an excellent way to illustrate the problems that arise when distributed users interact with each other. In a typical 3D-shooter game, a player enters a world model that is populated by monsters and other interactive players. Monsters are usually controlled by software residing on each player's local computer. Synchronizing the monster's actions with those of the player shooting at them is relatively easy and is not the topic of this article. However, it is much more challenging to synchronize information about all of the distributed players on computers around the world.

While running through a dungeon, your location and position (actually, the location and position of the graphic avatar that represents you in the game) are calculated by the simulation algorithms loaded on your local computer. That information is then transmitted via the network to all of the other players in the dungeon. Of course, it takes a small amount of time for that information to travel to all of those remote computers and for those computers to read the information in and change the position of your avatar. Imagine that your avatar is hiding behind a large crate and you order it to peek out from behind the crate and duck back immediately. The network message that moves your avatar out from behind the crate may travel to the other players very quickly. But the following message, the command to duck back, may be delayed along the thousands of miles of computer cable that make up the internet. This delay may be caused by variations in the processing load at any point along the way, increases in network traffic, failed computer equipment, or many other factors. And, because other players are often spread all over the world, the "duck back message" is usually delayed a different amount of time in reaching each of the other players' computers. In this situation, the avatar on your local computer may have only been exposed for 1 or 2 seconds. However, the avatar on all of the other computers will be exposed for a much longer period. It will be frozen in the exposed position because the "duck back message" has not arrived on those computers. Therefore, when your opponents look around the dungeon, they see your avatar standing frozen in plain view. Of course they unload multiple rounds from their plasma weapon into your character.

You were robbed – killed by a network lag. However, interestingly enough, when the plasma bolt from a remote shooter arrives at your simulation node, it informs you that you have been hit while standing in plain view, but that message arrives long after you have ducked back behind the crate. Your simulation algorithm has little recourse but to destroy your character even though

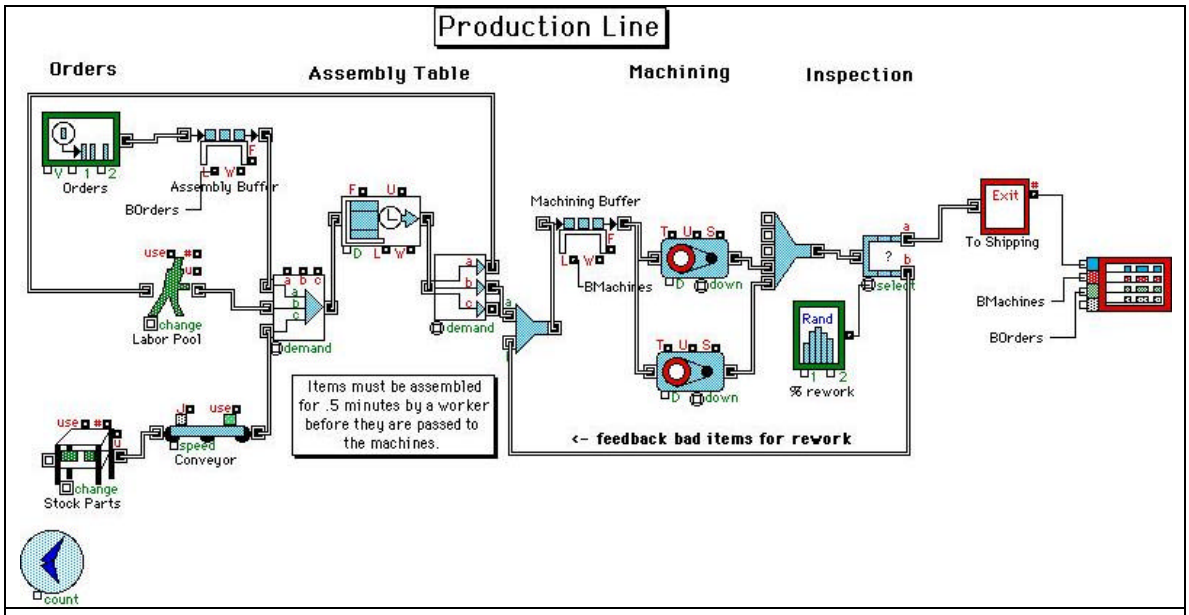
you are safely hidden. Failure to do so would create an inconsistency from the perspective of all the other players who see your avatar standing at the impact point of the plasma bolt.

Parallel and Distributed Simulation Technology. The example above illustrates one of the fundamental issues that arise when developing distributed virtual worlds or simulations that interact over a network. The delay imposed by the computer network interferes with the true pace of events and can corrupt the cause-and-effect relationships between these events. The problem is relatively new to the developers of consumer products like computer games and web-sites, but it has been around for several decades and scientists in universities and industry have been creating methods for addressing it. This type of research was initially focused on a class of applications known as discrete event simulations (DES), which are often models of factory production, seismic data analysis, laminar airflow studies, and visualizations of nuclear explosions. The techniques developed for those problems have been applied to distributed military wargames as well. Simulations of combat forces belonging to each military service were the first to implement some of the more advanced synchronization techniques in an interactive environment.

In this article, we will describe the leading techniques for synchronizing distributed virtual worlds and will provide examples of the algorithms that accomplish this. Readers who require a deeper understanding of any one of these techniques should consult the references listed at the end of the article.

DISCRETE EVENT SIMULATION

Since distributed synchronization techniques began in the discrete event simulation (DES) field, we should begin by describing the basic concepts behind DES and showing the similarity to virtual worlds that are just now emerging. A DES attempts to capture information about the real world in a form that can be used to study or illustrate its dynamic behavior. A photograph or 3D model of a factory is a static picture of that system. But a DES of a factory can capture the behavior of each machine and the cause-and-effect relationships between the machines when they are running.



Simulation of a Manufacturing Production Line
Courtesy of Imagine That Inc.

Fundamental Components

The fundamental pieces of a DES model are State, Events, Transition Functions, and a Time Advance mechanism.

State. The state of the system is simply a group of variables that describe the system at a specific point in time. In many ways the state is equivalent to a static photograph of the system. It captures all of the important variables, but can represent only one or a few values of these at any one time. The state variables of one avatar in a computer game may include its position, orientation, rate of movement, weapon list, ammo count, and health.

Events. Events are interactions that occur within the system that cause it to change the value of one or more state variables. These events enter the system from some external source. For a DES performing an analysis of a factory, events are often pre-loaded in the initialization data and read into the simulation when it is started. In an interactive environment like a game, events enter the system as a result of a player entering orders via a keyboard, joystick, or other control device. They can also be generated by automated behaviors of computer-controlled monsters in the game. Examples of events that change the state variables of a game avatar include Move, Duck, Pick-up Weapon, Fire Weapon, and Explosion.

Transition Functions. Events arriving in the simulation do not have a magic power to locate state variables and change them. The association between specific events and specific state variables is made by a set of transition functions. These are the dynamic modeling part of the simulation. A transition function may be contrived specifically to handle a one type of event. It understands the format and content of that event and the relationships to specific state variables.

This transition function calculates the type or magnitude of change that an event has on each state variable. A transition function known as "Movement" may be responsible for reading in a "Move" event, calculating its effects, and changing the position, orientation, and rate of movement of an avatar in a dungeon.

Time Advance. Events being processed in a simulation or virtual world must be sequenced according to some criteria. In most cases, this is done by the time at which the event was scheduled to execute. Therefore, some time advance mechanism must exist to move simulation time forward to allow future events to be executed. This is essential when ordering all the events from many remote sites. Because every simulation does not necessarily want to advance time in the same manner, there are several different types of time advance mechanisms.

Event Processing

During execution a simulation may generate hundreds or thousands of events. These queue-up to be executed in the appropriate order. In an interactive computer game, like a 3D shooter or a real-time strategy game, the computer may be receiving events from two or two dozen other players on the network. The software must attempt to order those events into a single list that is in the correct causal order. Causal order means that if a monster throws a fire ball at the player you are looking at, your simulation processes the movement of the fireball before it processes the destruction of that other player's avatar. It would be causally incorrect for the avatar to burst into flame before the fireball was even thrown.

Time-Stepped Simulation. For simulations that contain a large number of objects that are constantly changing state, it is natural to treat simulation time advance like the advance of a clock. The simulation clock ticks like a discrete watch and every object is updated to represent its state at the new time. Each game avatar moves to a new position, fires a weapon, takes damage from incoming weapons, etc. In a 3D shooter game these time-steps must be very small so the player does not see the virtual world jump from one position to the next. This is accomplished in the same manner that a movie is played. When a movie is run at 30 frames per second, the human brain perceives the series of pictures as a constant motion.

```
debtTime = 0;
while (!done) {
        startTime = getClock;

        <Do simulation processing here>

        elapsedTime = getClock - startTime;
        sleepPeriod = (stepSize/stepRate) -
                        (elapsedTime + debtTime);
        if (sleepPeriod <= 0) {
                debtTime = abs(sleepPeriod);
        }
        else {
                debtTime = 0;
                sleepFor(sleepPeriod);
        }
}
```

Event-Stepped Simulation. In some simulations events are so few and far between that time-stepping results in long periods of processing where nothing happens. Material in a factory may flow from one machine to the next every 20 minutes. Modeling this at 30 frames-per-second would be a huge waste of processing time and power. It would be better to queue each event and process them in causal order, but allow the time stamp on the event to set the simulation clock time. Simulation time would then jump from one event time-stamp to the next without representing all of the values in between. This can save a huge amount of processing and allow the simulation to finish in a much shorter period of time.

Event-stepping a simulation can also be used in interactive simulations with many events occurring simultaneously. When the simulation is structured this way, an additional mechanism must be implemented to notify objects when other objects have done something that is interesting to them. In a game, a remote avatar may run through the room your avatar is standing in. But unless your avatar has scheduled some form of event at that time, an event-stepped simulation will not pass thread of control to your avatar, and you will never see the other player run by or have a chance to react to him. To capture thread of control in this situation, your avatar must register some form of interest in objects that come near you. When this happens, the infrastructure managing the events will alert your avatar to the presence of the other player, giving you an opportunity to react to him. Though this mechanism works, it is rather cumbersome and can still miss notifying you of events that you are interested in. That is why most interactive simulations are structured as time-stepped simulations.

```
while (!done) {
        getNextEvent(smallest time stamped message in the queue);
        simTime = eventTimeStamp;

        <Process this event.  Send it to the appropriate object or event handler.>
}
```

Parallel and Distributed Simulation (PADS)

Like computer games, DES can be designed to executed on parallel computers or a network of distributed computers. These systems need to be synchronized for many of the same reasons that computer games must be synchronized, though the visual picture is seldom as easy to paint.

The PADS community has a rich history of inventing techniques to address the synchronization problem. For the last twenty years they have been solving problems that are only now beginning to emerge in consumer products.

DISTRIBUTED VIRTUAL WORLDS

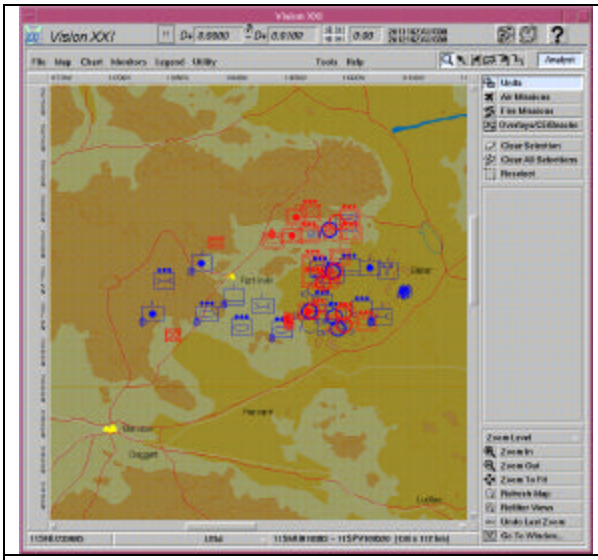
With the advances in computer simulation, graphics, and networking, technologies it is now possible to create distributed virtual worlds for the general consumer. These began as advanced experiments for military applications. Flight simulators were developed to train pilots in essential aircraft control and tactics for engaging enemy targets. These simulators were then linked together to allow two or more pilots to fly in tandem or to compete against each other in realistic, non-lethal training environments. But these connections were usually limited to a few simulators using a communications protocol that was unique to the operating environment of that system.

Networked Military Simulation

In 1983 the Defense Advanced Research Projects Agency began the Simulator Networking (SIMNET) project in which they created an economical way to link a family of tank simulators to allow multiple crews to train in the same virtual environment. The principles learned under that program became the seeds for much larger distributed interactive simulation technologies that emerged in the ensuing years.

Tank Simulator Computer Game
Courtesy of MaK Technologies, Interactive Magic, and Zombie Studios

The techniques were also applied to the wargaming community where they were used to join wargame simulations for military staff training. These wargames were the first to implement time synchronization techniques developed under PADS research projects.



Military Wargame Map
Courtesy of Tapestry Solutions Inc.

Computer Games and Web-sites

Today consumers are eager to play interactive 2D and 3D simulations of all forms of combat. Therefore, as the PC has evolved into a truly powerful computing platform, the game developers have adapted military simulation ideas into new types of games. It is now possible

to purchase a game form of every type of military simulation that was previously available only on powerful workstations and specialized image generators. In many cases, the visual and modeling capabilities of these games exceeds that of the best military systems.

In the past few years a huge market has developed for networked games that allow players to compete with each other from all over the world. These games face the same synchronization problems that have challenged parallel research projects and military applications for two decades.

As the World Wide Web continues to grow in sophistication, it will also evolve sophisticated synchronization requirements just as networked games have. Techniques for solving these problems already exist. In the following sections we describe the dominant methods that have emerged for synchronizing virtual worlds. Each of these can be applied to specific problems, but no single method is best for every problems.

INDEPENDENT OPERATIONS

Linking Diverse Military Training Simulators
Courtesy of Evans & Sutherland and Boston Dynamics

It is not absolutely necessary to implement a synchronizing protocol between distributed virtual worlds or simulations. In fact, many commercial and government products exist in which the events are processed independent from the processing being conducted on other computers. These have no concept of synchronization.

When allowing distributed applications to process independently, each event published onto the network arrives at the other simulators and is processed in a first-in-first-out order. This approach relies on the computer network, processing hardware, and operating system to deliver events in an efficient manner with negligible delays.

Slight improvements can be made by time-stamping every event and using that information to order events in the processing queues as they arrive. Since events are being processed FIFO and immediately, the advantage is only realized when events arrive faster than they can be processed.

Independent processing of events is usually implemented using the UDP network protocol. This eliminates the network traffic associated with acknowledging message receipt and calling for retransmission when a message is lost or corrupted. It makes more network bandwidth available for the transmission of events and object attribute updates.

Since each simulator is independent of all the others, there is no way to identify the difference between a simulator that is processing slowly and one that is having system problems that prevent it from updating its objects. An avatar in the virtual world may freeze in place for an inordinate amount of time. This could be caused by slow processing at the sender, network congestion, system failure, or a number of other issues. In the case of a system failure, the avatars from that simulator should be removed from everyone's virtual world. Otherwise the objects are not changed in response to explosions and other events that impact that object. To minimize this type of problem, the defense simulator community has implemented the concept of a "heartbeat". Each simulator is required to publish a state update message every five seconds. Since virtual world updates are usually much more frequent than that, in the neighborhood of 15 per second, even a slow processing simulator can achieve this easily. We can then deduce that avatars that are not updated within the five-second period must have experienced a system failure and should be removed from the virtual world.

In virtual simulators, the images on the screen typically update at least every second. This, along with the mandatory heartbeats, can create a very heavy traffic load on the network. To reduce this most simulators have implemented some form of dead reckoning (DR) on object movement. Given the last position, orientation, and velocity of a vehicle a remote simulator can extrapolate the position of that vehicle as long as none of those attributes change. Then it is possible for active objects to send much fewer messages about changes to their position. Dead reckoning can be done is many different ways, but three of the most common are given in the table below. Zeroth-order DR simply assumes that a vehicle remains in its previous position until told otherwise. First-order DR extrapolates the position of the vehicle based on its last known velocity and the elapsed time since the position was given. Second-order DR includes the acceleration rate of the vehicle in the extrapolation calculation.

Example Algorithm: DR equations

Zeroth-order DR:	DrLocation = lastKnownLocation;
First-order DR:	DrLocation = lastKnownLocation + lastKnownVelocity*timeElapsed;
Second-order DR:	DrLocation = lastKnownLocation + lastKnownVelocity*timeElapsed + (1/2)*lastKnownAcceleration*(timeElapsed)**2;

The independent operations described in this section were developed by the military simulator community and largely defined under the SIMNET and Distributed Interactive Simulation (DIS) programs. More details on the implementation of these can be found on the web site of the Simulation Interoperability Standards Organization (SISO) –http://www.sisostds.org/.

CONSERVATIVE SYNCHRONIZATION

In some applications it is essential that the distributed pieces of the system be strictly synchronized to allow them to vary the rate of progression of time, achieve identical event ordering, and stop or restart the simulations in a coordinated manner. These types of simulations often require regularly saving the state of the simulation and restoring that state in a synchronized manner. Wargames are also able to progress forward at different rates, such as twice the speed of real-time or half real-time.

"Conservative synchronization" is an umbrella term for techniques that keep all of the processes causally linked at all times. It may be implemented with many different structures, to include:

- Master Clock,
- Token Passing,
- Client/Server,
- Chandy/Misra/Bryant Algorithm, and
- Aggregate Level Simulation Protocol.

Master Clock. Synchronization can be accomplished by assigning one of the distributed simulations as the owner and controller of the clock. That simulation processes its own events and moves the clock forward, sending its time to all of the other simulations before it processes its own events. Those simulations accept that time and process events accordingly. Events that arrive from around the network are held in a queue until the master clock indicates that they can be processed.

Simulations that are slaved to the master clock are expected to operate fast enough to remain apace of the clock owner. As long as the slaved simulations run at least as fast as the master this approach works fine. However, if the shared simulation time must be mediated by all members of the virtual world, a more advanced mechanism is needed to keep the systems synchronized.

Example Algorithm: Master Clock Synchronization

```
Clock Owner:
while (!done) {
        simTime = incrementTime(prevSimTime);
        sendTime(simTime);

        startTime = getClock;

        <Do simulation processing here>

        elapsedTime = getClock - startTime;
        sleepPeriod = (stepSize/stepRate) -
                        (elapsedTime + debtTime);
        if (sleepPeriod <= 0) {
                debtTime = abs(sleepPeriod);
        }
        else {
```

```
                debtTime = 0;
                sleepFor(sleepPeriod);
        }
}
```

Clock Slaves:

```
while (!done) {
        simTime = waitTimeUpdate();

        <Process events at this sim time.>
}
```

Token Passing. In some situations, permission to publish and process events can be controlled by exchanging a network token. As an example, synchronization in an online, four-hand, bridge game can be controlled by passing a token from one player to another. Information about each card played is published to all of the other players allowing them to see what is played. But no player is allowed to play a card until they receive the network token that is cycling between all the players.

This method is very useful for turn-based games in which a fixed number of players join a game and remain in the game from beginning to end. A game server usually exists to match players together to begin the game, but the server does not have to play an active role in the progression of the game. If the game is one in which players will join and leave as it executes a server must be involved to accommodate this. It requires that the server receive control of the token at the end of a cycle of play or between each player's turn.

Most turn-based card and strategy games use a server to keep statistics and ranking boards on each player. The server contributes a certain amount of community to the game by preservig past events, ranking players, identifying future events, etc. But the primary objective in keeping the server involved in each game is financial. It is an opportunity to charge players a monthly fee for being part of the game community they have chosen.

Client/Server. Combining the ideas of Master Clock and Token Passing, Client/Server synchronization pulls more control of the distributed execution into a centralized server. Most networked computer games are implemented in this way. It provides much more accurate control of the distributed environment by releasing events to clients only when they are executable. The server can progress simulation time in accordance with a real-time clock or by some other controlled rate of advance. Most game servers progress according to a real-time clock, expecting each client to process events fast enough to remain in pace with this.

In addition to synchronizing time and events, the server can adjudicate situations in which network lag has created disjoint events like those described in the opening example of this article. The server can determine whether a weapon-hit should have happened in a perfect, non-delayed network environment. It can also make more sophisticated judgements about whether legal events should be allowed to happen based on how they will appear to the players

in the virtual world. The virtual world on the server is the most accurate picture of the state of the world. Each clients has a slightly inaccurate, reflected picture of the world to work from.

Chandy/Misra/Bryant (CMB) is the common name for a popular method of imposing event synchronization in parallel and distributed DES. Each of the authors was responsible for contributing concepts to the complete implementation of the technique. Since it was developed is a DES environment, it is optimized for an event-stepped world in which events are relatively sparse within a given time period. These events are usually loaded from an initialization file or are triggered by these initial events. They are seldom received from an interactive user of the simulation.

The CMB algorithm is implemented within the infrastructure software that manages the event queues and releases events to models based on the progression of the simulation time. CMB is a key part of determining the rate at which distributed simulation time progresses. Each event posted to the simulation infrastructure by a model is sent to all of the other simulations in the parallel or distributed environment. Upon arrival, the infrastructure places each event in a local queue that is identified as belonging to the remote simulation that generated the event. Therefore, the infrastructure on each distributed machine is holding multiple queues with events segregated according to their originator. These queues can be used like a scoreboard to determine where each distributed piece is in its execution.

The simulation infrastructure orders the events in the queue from the earliest to the latest. It then locates the event with the smallest time-stamp across all of the queues. This event is safe to process because we are certain that none of the distributed simulations will generate a new event with a time-stamp smaller than this one. Executing it can not lead to a cause-and-effect error with later events in the queues.

Each simulation in the distributed environment proceeds in the same manner, generating new events, posting new events, and processing the smallest timed event in its local queues. However, it is possible for a simulation node to process all of the events from one of the queues, leaving it with no knowledge of the time at which the remote simulation associated with that queue is processing. When this happens it is not possible to select the smallest event in the queues because the next event that arrives for one of the empty queues may have a smaller time that the time of other events that are available. Proceeding with the selection of an event to process would be causally dangerous. You may arrive at a point in time ahead of the time-stamp on the next event that arrives from that simulation.

Several solutions to this problem have been considered. If we assume that the empty queue should retain some memory of the time stamp of the last event that was removed from it, this may help the situation. This could then be used when the smallest time-stamp is determined. Unfortunately, this does not help because the memory of the time on the last event processed does not move forward until a real event arrives to fill that queue. Complicating this situation, it is possible for each simulation to empty their event queues in an order such that they all become

deadlocked. Imagine a 3-computer environment with the simulations labeled **A, B**, and **C**. Simulation **A** may have emptied the queue of events sent by simulation **B**. **B** may have emptied the queue from **C**. **C** may have emptied the queue from **A**. In this situation, each simulation is waiting on one of the others in an order that deadlocks the entire distributed environment.

To solve this, we need another piece of the CMB solution. Each simulation generates "null messages" or "null events". These do not carry any information about actual modeled events, they simply carry the simulation time of the next event that the simulation intends to generate. Therefore, when one of the nodes reaches a point where it is not going to generate events for some time it creates a null message that tells others where it intends to go next. Many different methods can be used to determine when a null message needs to be sent by the software. An ideal method must be independent of the specific models running on the infrastructure. It must be something that will work in all situations. The most generic and failsafe method for doing this is generating a null message after you process each real event. The null message then acts as a pre-announcement of the time that will be on the next event when it is finally generated. One drawback to this method is that it generates many more null messages than are actually necessary. Another approach would be for a simulation to generate a null event each time it sees that one of its queus is empty. This minimize null messages and insure that deadlock does not occur. But it may allow remote simulations to sit idle for some long periods of time.

There is another problematic question to be answered. How does a simulation know what the time stamp will be on the next event it will generate? If the simulation is time-stepped, the next time stamp is equal to the current time plus the size of one step increment. If the simulation is event stepped, the next event could be at any time in the future. Luckily, even event-stepped simulations are written with some understanding of their inherent step-size. It is possible for an event stepped simulation to identify the soonest it is possible for the software to generate the next event. This is usually based on specific details about how the simulation is structured. Therefore, after generating all events for time t, the next event must be generated at time $t+1$ (or more strictly $t+dt$). The time step size or simulation clock counting increment dt is defined to be the "lookahead" of the simulation. The lookahead value is then added to the current time to determine the time stamp that will be delivered in the null message.

As the result of experimentation it was apparent that the rate of progress of distributed virtual worlds was directly effected by the size of the lookahead of the simulations involved. Larger values allowed the federation of simulations to move ahead more efficiently than smaller values. But large values also reduce the flexibility of the simulation algorithms. Therefore, the selection of lookahead is always a trade-off decision between modeling flexibility and processing efficiency.

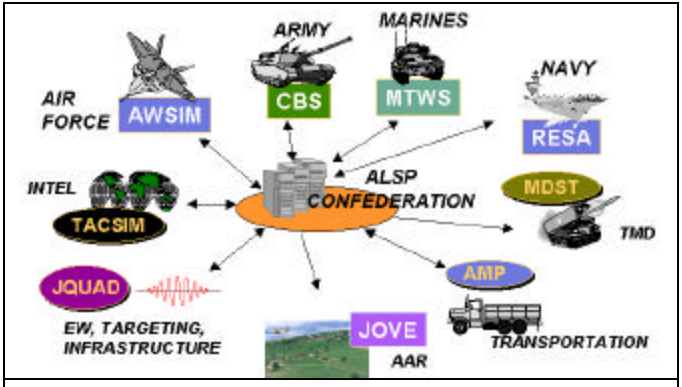
Example Algorithm: Chandy/Misra/Bryant Conservative Synchronization

```
while (!done) {
        waitUntil(each FIFO queue contains at least one event);
        getEvent(smallest time stamped message in all queues);
        simTime = eventTimeStamp;

        <Process this event>

        sendNullMessage(time stamp = simTime + lookahead);
}
```

Aggregate Level Simulation Protocol. In the early 1990's the military was searching for an alternative method for synchronizing distributed wargames. They had been using the Master Clock method and were suffering from some of its limitations. The CMB synchronization algorithm was considered and experiments conducted. The time-stepped nature of the wargames indicated that some very serious performance improvements could be made for the specific Joint Training Confederation (JTC) that was being created.



Linking Military Wargames.
The Joint Training Confederation.

Since each simulation was time-stepped, the lookahead value was very obvious. However, each simulation was responsible for many hundreds or thousands of objects and each of those executed and generated events at each time-step. Therefore, a pure CMB algorithm would generate a null message after each real event, essentially doubling the number of event messages being sent through the network. Since several hundred units all executed one or more events at the same time stamp, the information carried in the null message was very redundant. Therefore, a modification was devised that eliminated thousands of null messages, but retained the power of the CMB algorithm.

The infrastructure was modified so that it no longer considered the time-stamp on each event in its search for the lowest distributed simulation time. Instead it would consult only a new, special type of event – the "Advance Request" event. Before stepping to the next time-step, each wargame would post an "Advance Request" to the infrastructure. These would be exchanged

between all members of the virtual world and evaluated to determine the lowest time that was being requested from all of the simulations. Each instance of the infrastructure would then turn around and give its simulation an "Advance Grant" for the smallest time. Simulations that wanted to advance to that time would do so. Others would wait for an advance grant for the time they had requested. Those that did have events for the time granted would process them and post another "Advance Request" for a time further in the future than the last one. As this continued simulation time would advance, each simulation would be able to process its events, and each simulation would have control over the rate at which the distributed time progressed.

This approach to event management and synchronization became part of the Aggregate Level Simulation Protocol (ALSP) that is used to tie distributed wargames together. More information on this protocol and project is available at http://ms.ie.org/alsp/.

OPTIMISTIC SYNCHRONIZATION

In an effort to squeeze even more speed and efficiency out of distributed simulations, a group of scientists began experimenting with ideas in which the simulations are free to process events as fast as possible. But they are still required to retain the cause-and-effect relationships between events and objects spread across the distributed simulation processes. This was pioneered by David Jefferson at the University of Southern California. It was clear that every implementation of conservative synchronization required faster simulation nodes to remain idle while waiting for one of the slower members to process events. This was wasting CPU cycles that might be used to process events locally. In fact, processing these event "ahead of time" might allow the entire simulation federation to finish its job much quicker. Unfortunately, processing events ahead of time may break the cause-and-effect relationship between events and objects on different computers. Some method was needed to use the available CPU cycles, progress simulation time efficiently, but retain cause-and-effect relationships. Thus was born Optimistic Synchronization.

We should make it clear that these ideas were being formulated for DES used to analyze data and process events that were pre-defined in the computer model. Interactive users were not involved during the execution of those simulations. We will add interactive users to the mix after we have described how Optimistic Synchronization works for non-interactive simulations.

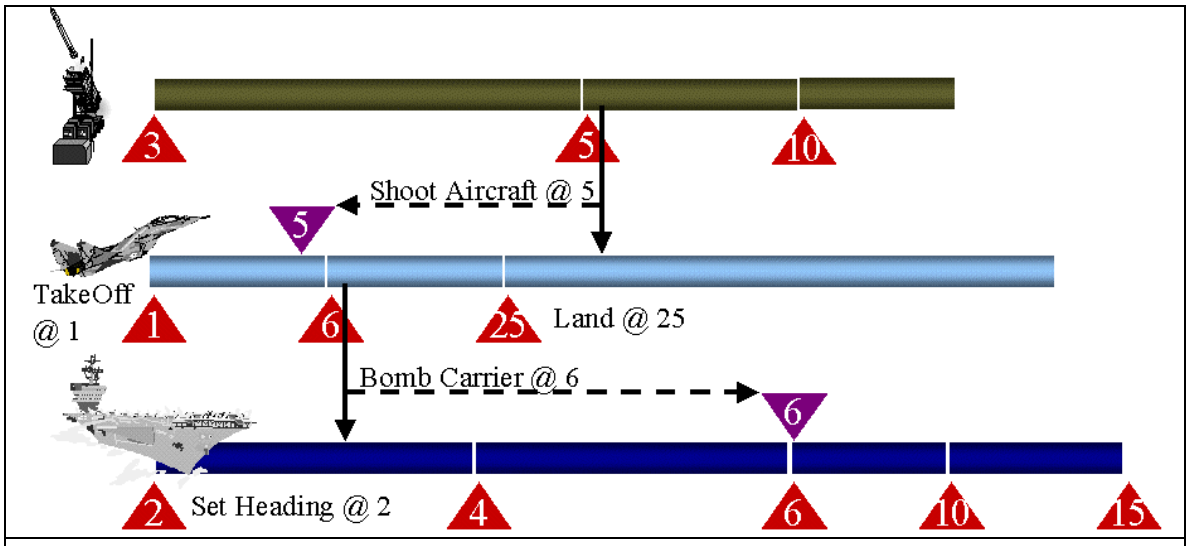
Time Warp. David Jefferson created an approach called Time Warp that implemented his ideas for optimistic synchronization and which has become the foundation for most of the work in this area. Under Time Warp, each simulation on the network operates as if it is running entirely by itself. Every event that arrives on the computer, whether from a local data file or a network message is delivered immediately to the simulation models for execution. The models are assuming that both the local and distributed events will arrive in the correct time-stamp order. If this assumption were 100% correct, then the cause-and-effect relationship would not be violated and maximum use would be made of the CPU cycles available. However, this is assumption is never 100% correct. In many cases the events do arrive in the wrong order and

events are processed incorrectly. When this occurs, some mechanism must be used to redo the calculations in the correct order. The "undo" and "redo" operations required to accomplish this will require CPU cycles. Therefore, Time Warp is a balancing act between the CPU cycles captured for processing events early and the CPU cycles lost when fixing problems caused by this approach.

We will explain the operations of the Time Warp mechanism with a combat example. Imagine a federation of three simulations. One handles all army units and we will call it Ground. Ground is responsible for tanks, soldiers, trucks, and surface-to-air missile batteries. The second handles all airborne units (Air) – fighters, bombers, tankers, electronic warfare, and AWACS. The third handles all sea-going units (Sea) – carriers, battleships, submarines, and supply ships. In a particular scenario these three simulations are using the same computer hardware, but each has a very different software model and must manage widely varying numbers of units. Ground must simulate the operations of 10,000 units, Sea must simulate 1,000 ships, and Air must simulate 100 aircraft. In this situation it is almost certain that the three CPU's will have very different loads imposed on them and will progress forward at different rates.

The Air simulation launches a flight of aircraft to bomb a carrier at sea. That simulation flies the planes to the projected (via dead reckoning) position of the aircraft carrier and drops the bombs on that location. However the Sea simulation has not processed to that point in time yet. So it accepts the posted bombing event and places it on the queue for future processing. The Air simulation continues to process, flying the aircraft back to the base and landing them. At some point, Sea arrives at the simulation time of the bomb release, recognizes that the bombs were dropped on the correctly predicted location of the carrier and processes the explosion and resulting damage. This is a best case scenario for the Time Warp mechanism.

However, assume that the Ground simulation has been processing through time very slowly. Long after the bomb has been dropped, the Ground simulation processes the flight event that took the aircraft past a surface-to-air missile (SAM) battery on its way to the target. The SAM model recognizes that it can detect the aircraft, shoots a missile at it, and hits it. This all occurs at a simulation time prior to that at which the bomb was dropped. When the shoot-down event is posted to the network and arrives at the Air simulation, the Air simulation must "undo" all of the events it executed after the shoot-down time and recalculate the sequence with the shoot-down event inserted at the appropriate time.



Rollback in a Time Warp Simulation.

To support this, Time Warp and Optimistic Synchronization methods must record all changes to state variables. These records are then reapplied to the aircraft object in reverse order until it is "rolled back" to a state prior to the shoot-down event. The Air simulation then inserts the shoot-down event in the event queue and reprocesses the sequence. This time through the aircraft discovers that rather than having a successful mission that can be told around the officer's club, it is shot down, the pilot ejects, and must be rescued before he is lost at sea. Since the aircraft was rolled back and shot down, there must also be a mechanism for retracting events that were generated in the first pass, and must be undone in the second pass. Events that were generated by the aircraft are stored, just as state changes were, so they can be retracted (rolled back) as well.

Object state rollback is relatively easy because it is entirely contained on the local computer. But retracting events posted on the network is accomplished by posting "anti-messages" or "anti-events". These travel to the remote nodes and are used within each instance of the infrastructure to locate the original event and remove it from the queue. If the original event has already been processed, the anti-message will cause that simulation to rollback as well. It must rollback and reprocess its sequence of events with the retracted event removed from the queue. When the rollback of one simulation causes another to rollback as well, it is referred to as "cascading rollback". As the name implies, in some situations the cascading effect can go much further than just one or two objects. It may trickle all through the entire distributed simulation.

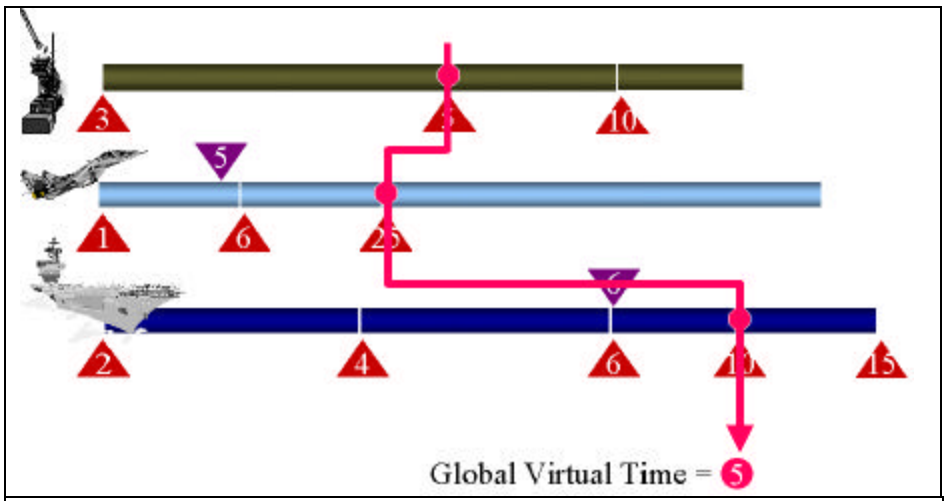
As chaotic as this sounds in human terms, within the computer it is merely an annoyance that costs CPU cycles and prevents the parallel or distributed simulation from finishing as soon as was hoped. The processing of events, anti-events, state changes, and rollbacks are just computer operations designed to study a problem and arrive at an answer as soon as possible. There is no need for an operator to see this non-monotonic progression of time. There is one major fly in the ointment - the state and event data that is stored in the computer in preparation

for rollback consumes valuable RAM. Some mechanism is necessary to release this information when it is no longer needed and reclaim the memory it is using. To do this we must be able to select some simulation time beyond which roll back can not occur. This is where the CMB and ALSP concepts of a unified federation time are valuable.

If we evaluate the current processing time of all of the distributed applications, one of them has the smallest simulation time-stamp. That time represents a point at which no additional events can be generated with a smaller stamp. Therefore, rollbacks can not push any member of the federation back further than that time. Any saved state and event information prior to that time can be safely reclaimed and used for other operations (perhaps to store the newest state saves that are being generated).

Unfortunately, because this is a distributed simulation there may be many events that are in-transit between the sender and the receiver. This requires the use of some rather intricate algorithms to identify the true smallest simulations time in the distributed world. Descriptions of several of these algorithms are beyond the scope of this article and can be found in Fujimoto's book. This lower time boundary is known as the Global Virtual Time (GVT). GVT can be viewed as the unified time of the distributed simulation.

There are additional details on the implementation of Time Warp and Optimistic Synchronization that are beyond the scope of this article. Interested readers should consult *Parallel and Distributed Simulation Systems* by Richard Fujimoto. It is the definitive source of information on this topic.



Identifying Global Virtual Time

Man-in-the-Loop Time Warp. Early in the discussion of Time Warp we pointed out that it had been developed for analytical applications that did not include interactive man-in-the-loop operators. Some work has been done to apply this technique to interactive simulations and

gaming environments. But, in general, the insertion of an interactive user negates most of the advantages of this approach. Human players expect the simulation to progress at real-time, so the advantage of processing events as fast possible is reduced to remaining synchronized with real time. But since the simulation is not allowed to fall behind real time at any point, the computer hardware must be capable of processing all events and rollbacks at a real time rate. This may require more powerful hardware than is necessary for conservative synchronization.

Interactive players also can not witness the pre-processing and rollback of events in the simulation. The simulated world must be revealed to them only as events are guaranteed not to be rolled back. Therefore, all interactive users must "ride GVT". Meaning that they see only events and states as GVT advances over them. Since the player may insert new events at any time, he becomes the default slowest simulation in the federation. The progress of GVT is totally governed by the progress of the players, which is defined to be at real time.

The one remaining advantage of implementing Time Warp in interactive simulations is that it allows processors to work on large clusters of events before the player sees them. This has the potential of avoiding simulation slowdown at critical times when lots of events occur simultaneously. This work may be entirely pre-processed by the Time Warp simulations. However, it is probable that during large clusters of events the interactive player will want to insert new events reacting to what he sees happening. This will cause rollback at the worst possible time.

In general, Optimistic Synchronization is not the best solution for interactive simulations used for military training or computer gaming.

RECOMMENDED APPLICATIONS

After describing many of the synchronization methods used for distributed virtual worlds, it is appropriate that we identify which are best for different types of applications. This list can not be all encompassing, but it can categorize many of the major types of simulations.

Synchronization Method	Application
Independent	Flight Simulators
Token Passing	Card Games Turn-Based Strategy Games
Master Clock	Model-to-GUI Coordination
Client/Server	3D Shooter Games Real-Time Strategy Games
CMB/ALSP	Distributed Wargames
Optimistic	Large Analytical Simulations
Regulated Optimistic	Analytical Simulation with High Interactions

EXCELLENT REFERENCES

The following books are the very best references for information on synchronizing distributed virtual worlds. Fujimoto's book describes the techniques that have been developed in research laboratories and those that have been applied in fielded applications. The pseudo-code examples contained in this article are derived from information in Fujimoto's book. Kuhl's book describes a specific distributed simulation infrastructure, the High Level Architecture, which has been developed for military simulations. Singhal gives an excellent background of the evolution of distributed virtual environments and describes the environment that has been created at the Naval Postgraduate School. The Game Developers Conference is the premiere event where computer programmers gather to exchange their techniques, including those on networked multi-player games. The SISO and ALSP web sites provide useful documentation on DIS, ALSP, HLA, and related topics.

APPLICATION	REFERENCE
Parallel Discrete Event Simulation	Fujimoto, R. 2000. *Parallel and Distributed Simulation Systems.* New York: Wiley-Interscience.
Distributed Military Simulation	Kuhl, F., Weatherley, R., and Dahmann, J. 2000. *Creating Computer Simulation Systems: An Introduction to the High Level Architecture*. Upper Saddle River, NJ: Prentice Hall PTR. Singhal, S. and Zyda, M. 1999. *Networked Virtual Environments: Design and Implementation*. New York: Addison Wesley. Simulation Interoperability Standards Organization. Orlando, Florida. http ://www.sisostds.org/ Aggregate Level Simulation Protocol. http://ms.ie.org/alsp/.
Computer Games	Miller Freeman. *Proceedings of the Game Developers Conference*. San Jose, California: Miller Freeman. http://www.gdconf.com/.