Time and Event Synchronization Across an MMPG Server Farm

Roger Smith and Don Stoner – Modelbenders LLC
gpg @ modelbenders.com

Abstract

The server farms that support massively multiplayer games contain hundreds of computers, many of which represent geographic areas that are so close or intertwined that the operations on these machines need to be tightly synchronized. As the richness of game content and cross-server interactions increases, the need for synchronization grows as well.

In this chapter we describe algorithms that have been developed for parallel and distributed simulation systems to provide guaranteed synchronization of event execution and time advance across any number of networked computers. Modifications have emerged which customize the original algorithms to make them more efficient for interactive virtual worlds.

The goal of this chapter is to describe the algorithms in terms that can be understood by an experienced programmer, provide the necessary computer code to begin implementing the techniques, and explain the costs and benefits of using these techniques.

Introduction

Synchronization of event execution and time progression across parallel and distributed computers has been an issue in high performance computing circles for decades. The programmers who created massive models of nuclear blast effects found that their simulations were too big to be handled by a single processor of any size. Therefore, they turned to parallel computing to give them the horsepower to run these in a reasonable amount of time. But that immediately created a new problem of synchronizing the events that were occurring on these multiple independent processors.

Since then, distributed computing has spread to a number of other application domains. Interactive training for the military and computer gaming are two of the most popular. Massively multiplayer games (MMPGs) are the most extreme form of distributed computer game and demand event synchronization just as the nuclear models and training simulations do.

Impacts on the Massively Multiplayer Experience

MMPGs like Asheron's Call find that event and time synchronization across machines in their server farm is a major issue. Jeff Johnson of Turbine Entertainment says that, "the biggest problem that Turbine has in managing its seamless worlds' server-side is in

dealing with asynchronicity and serializing server game state at arbitrary points of execution" [Johnson04]. Generally MMPG server systems follow one of two major architectures: (1) the zone-based model where objects are represented on a single machine that is responsible for a specific geographic area of the virtual world, and (2) the distributed (or seamless) model in which objects are represented on multiple servers and their state values and event lists must be constantly synchronized [Beardsley03]. Though the second model desperately requires a reliable synchronization mechanism, even the zone-based model contains game events, boundary conditions, and system management operations that require synchronization.

Poor or non-existent synchronization across servers impacts both the players' experience with the game and the ability to manage and control operations within the game engine. In some cases, inconsistent event execution can result in different outcomes of event sequences on the servers and potentially expose this inconsistency to the player client machines [Smith00].

Synchronization is a difficult thing to achieve and does impose performance costs and operational limitations. However, the increasing complexity of MMPGs and the growing horsepower and bandwidth that drives them is going to justify the resources for synchronization just as these resources have opened the door for better AI and physics in the past.

Available Solutions

There are a number of different solutions to the event and time synchronization problem. Each provides slightly different capabilities at different performance costs. The first, most common, and least expensive is the *best effort* method. This calls for each message receiving process to buffer messages, order them according to their timestamp, and execute them in the hope that all of the sent events for the buffered period have been received. When a late message is received, the event managing software must make a decision to either execute the event late or to delete it. This decision is usually very specific to the game and the type of the event. It is essential that this decision is made deterministically and applied identically on every machine. However, even a deterministic algorithm cannot deliver uniform results on multiple machines. Message delivery delay varies from one machine to the next and the buffering of events does not result in the same events being included in the buffers on every machine. For example, consider an event E1 sent to computers M1 and M2. On computer M1, event E1 may have been received, ordered, and properly executed. But on computer M2, event E1 may have arrived late and been subject to either late execution or deletion. This uncertainty of results is a major motivation for the creation of more predictable synchronization algorithms.

A second popular method of synchronization is through the use of a *central timeserver*. One process is anointed as the master of all time progression. Its job is to set the pace of execution for all game processes in the server farm and to determine when conditions warrant moving forward, slowing down, or stopping. This method improves on *best effort*

in that all of the processes are slaved to one master and thus remain much more closely aligned in time. However, it does not provide any mechanism to guarantee that messages are executed in the same order on multiple machines. Also, as the "master of time", this process can ignore the performance issues of heavily loaded slave processes, allow them to fall behind, and create opportunities for causal event violations. Many of the message passing algorithms that have been created to address this problem are actually ad hoc or partial implementations of synchronization algorithms we are about to prescribe.

The leading method for reliable event and time synchronization is the *Chandy/Misra/Bryant* (CMB) algorithm and several useful modifications of this algorithm. CMB requires exchanging messages between servers that define which event timestamps have been executed and determine the readiness to move forward to the next time increment [Fujimoto90]. In this chapter, we will describe CMB and some modifications that are particularly useful to MMPGs.

Another synchronization method that is very popular within academia and some analytical communities is Time Warp. This is a very exotic method that is difficult to understand, implement, and modify for an MMPG. Readers interested in this technique should dig into the references provided at the end of the chapter [Fujimoto00 and Smith00].

Time in the Virtual World

Before explaining CMB event and time synchronization in greater detail, it is important to establish some basic properties of time management in simulations and games. Some of these characteristics are required to create a simulation that is causally consistent and others are necessary to enable CMB to work.

1. Virtual Time is Real. When discussing the subject of time in a virtual world, the significant value is the time that is created and managed by the software, not the "real time" experienced by flesh-and-blood players. When we manage time advance in a game we are often attempting to align virtual time with real time, but all events and the entire digital world are referenced to virtual time, not real time.
2. Discrete Step Size. In a game, time moves forward in discrete increments. In many systems these steps are so small that it appears to the player that time is moving continuously. But that is an illusion just as a movie appears to present a continuous moving image even though it actually has a step size of 24 frames-per-second. Effectively, the step size defines an increment that is small enough that all events scheduled in that period can be treated as if they occurred simultaneously.
3. Monotonically Increasing. Game time is always monotonically increasing. This means that event timestamps always increase or stay the same. They do not ever go backward. This is an important property because it means that if a process generates an event with a timestamp of 100 on it, then it will never again generate an event with a stamp of 99, 98, or any other value less than 100.
4. Event Timestamps. All events in a simulation are time stamped. There are no orders, commands, inquiries, or reports that are created without a stamp indicating

the time at which they should be executed. It is not so important to stamp them with the time that they are created, though there is some value in that as well, the essential time is that at which the event must be executed by the game.

5. Network Message Lag. The delivery of messages across a computer network always takes time and induces lag. Therefore, messages on the receiving end are "old" in that they represent the state of the sender some delta milliseconds in the past. Additionally, there is no guarantee that messages sent will be received in the order that they were sent, or even received at all.
6. Limited Remote Information. The receiver of messages always has a limited amount of information about the state of the sender of the messages. This limitation has a direct impact on the content of messages and the reasoning that must be performed on the receiving computer.

Managed Synchronization

Chandy/Misra/Bryant Algorithm

CMB is known as a "distributed k-reduction algorithm". It can be used to synchronize any number of independent processes running on different processors or on the same processor. Using event timestamps from all of the participating processes the CMB algorithm calculates the "Global Virtual Time" (GVT) for the entire group. GVT is the minimum timestamp on all exchanged messages. Virtual worlds generate and transmit events that are scheduled to be executed at some time in the future. GVT identifies the latest timestamp on events that can safely be processed without creating a causal error. All events up to and including those at GVT can be processed without worry that a synchronized process will create another event in the past of GVT.

When implementing CMB, the infrastructure that receives the event messages maintains one queue for each of the other remote participants in the synchronization. As events arrive from the remote processes, they are logged in the appropriate queue (Figure 1). The GVT mechanism evaluates the timestamps in each queue and identifies the lowest value in the queues. That timestamp becomes the next GVT value and all events with that timestamp are released to the modeling software for execution. Events with higher timestamps remain in the queue awaiting future release. As events are released, it is possible for one or more of the queues to become empty. When this occurs the mechanism cannot advance the value of GVT because it cannot determine what the lowest timestamp for the associated process will be. Therefore, when this occurs, GVT must remain at its current value until an event arrives to fill the empty queue. This algorithm is illustrated in the following code sample.

```
/* basicGVT illustrates the original
Chandy/Misra/Bryant algorithm – without mods */
void basicGVT( void )
{
/* integer used for iterating through the list of
processes */
```

```
int processIndex = 0;

/* integer used for iterating through the event queue
of each process*/
int queueIndex = 0;

/* Initialize stop */
     bool stop = NULL;

/* Next Global Virtual Time “nextGVT” initialized to
some large number */
nextGVT = setNextGVT (99999);

/* This while loop iterates through each of the
processes and their queues, and checks for the lowest
timestamp.*/
while ( ((processIndex < numberOfProcesses) && (stop
== NULL)) ; processIndex++)
{
     /* If event queue is not ordered, check every
event. If ordered, just check the first event. This
for loop iterates until the queue is empty */
for (queueIndex = 0; e[processIndex][queueIndex];
queueIndex++)
{
/* Checks the timestamp to see if it is the new
minimum and sets nextGVT if it is. */
nextGVT = min(nextGVT, e[processIndex][queueIndex]-
>timestamp);
     } /* end for */

/* If the current “processIndex” queue was empty. */
     if (queueIndex ==0)
{
/* stop calc*/
          stop = 1;

/* Sends a NULL message.  Null messages are required
to keep the process from entering a deadlock state. */
          sendNullMessage ( );

     } /* end if */

} /* end while */

if (stop == NULL)
{
```

```
        GVT = nextGVT;
    } /* end if stop */

    } /* end basicGVT */
```

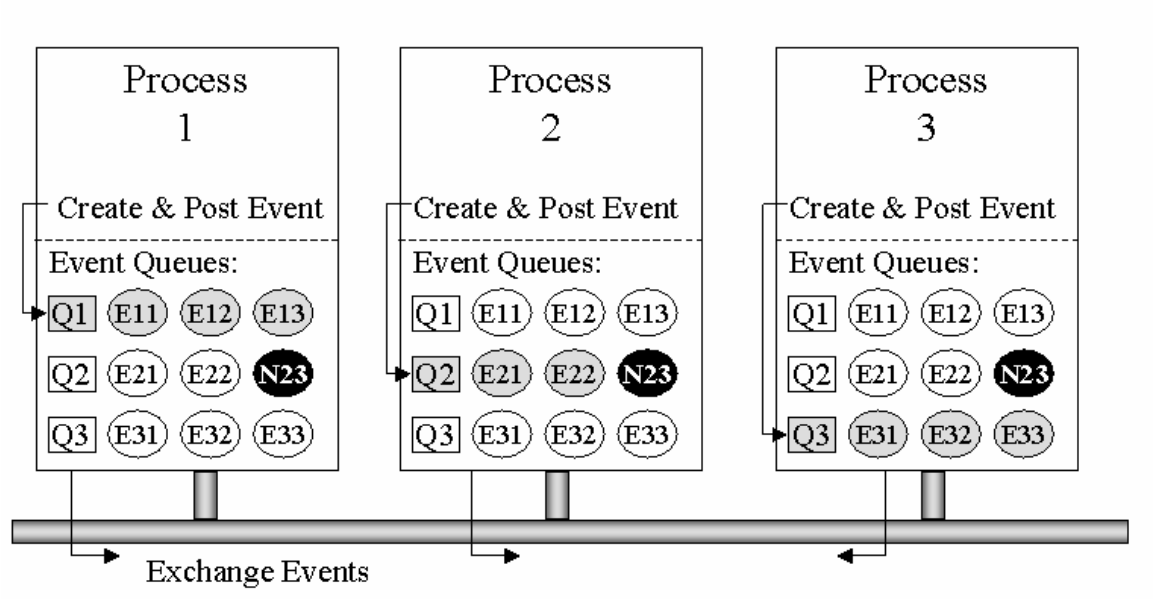

Figure 1. Chandy/Misra/Bryant Event Queues to Calculate GVT

Empty queues can result in a deadlock in which process P1 is awaiting an event from process P2, while P2 is waiting for an event from P3, and P3 is awaiting an event from P1. To break this deadlock, CMB implements "Null Messages" (the black messages in Figure 1). These are not true executable events. They are messages that simply carry the timestamp for the next event that a process intends to generate. Null Messages are usually generated at a scheduled rate that is driven by the largest acceptable deadlock time. The original CMB algorithm generated a Null Message after each real executable event. This essentially reduced deadlock time to zero, but at the cost of doubling the number of messages being sent between computers. The newer timed-release mechanism is much more bandwidth economical, but at the expense of a short deadlock period.

The computation of GVT is so simple that the CPU expense is almost insignificant. The real impact is that it regulates the pace of all processes to match that of the slowest process in the synchronization. This is one of the features that later modifications have improved upon.

Advance Request/Grant Modification

CMB was designed for analytical simulations such as nuclear blast studies or models of national air traffic patterns. When this algorithm migrated into the military training domain, specific modifications were made to improve its performance. In a military training simulation, there can be thousands of objects sending event messages to dozens

of different computers. Evaluating the timestamp on all of these messages proved to be redundant and unnecessary. During a given time step, thousands of objects generate event messages with the same timestamp. Under these conditions, the GVT algorithm found itself comparing thousands messages with the same timestamp.

To reduce these comparisons, an "Advance Request" and "Advance Grant" message pair was created. Advance Request was a request by the process to move to a specific time in the future (Figure 2). In most cases, this corresponded to the timestamps on the thousands of event messages. But, it reduced the number of timestamps to be compared for GVT from a number on the order of the number of objects in the virtual world (n*10,000s), to a number on the order of the number of processes running (n*10s). In the case of large object database this can represent an improvement of three orders of magnitude. When the GVT algorithm determines the next safe timestamp to advance to, it provides an "Advance Grant" message to those processes that are allowed to move to their requested time. Processes that had requested a time further in the future do not receive a response and are expected to wait until they receive a grant – usually after a slower machine has caught up to that time.

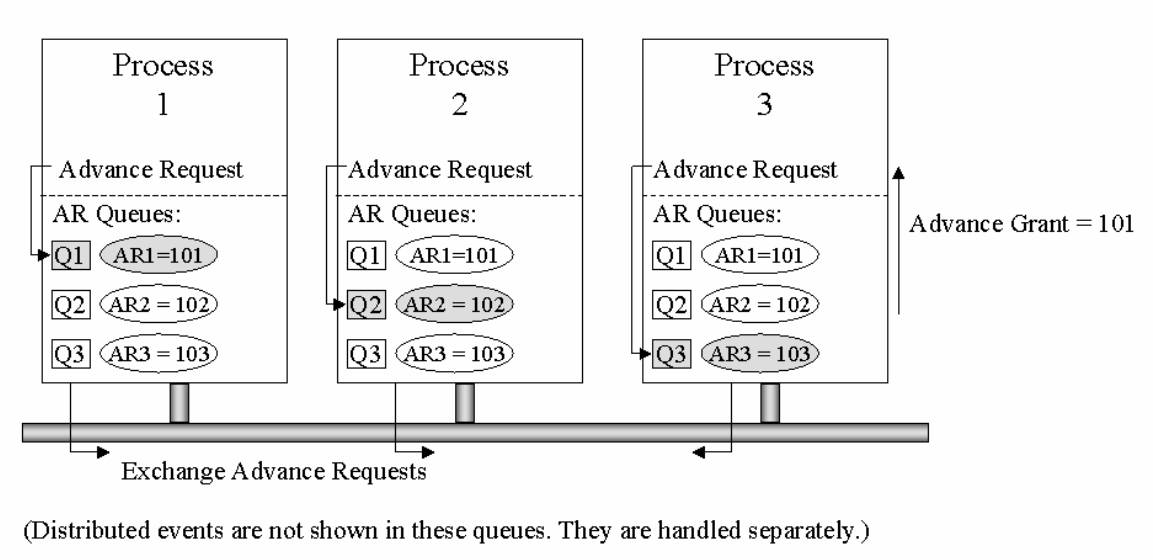

Figure 2. Advance Request Messages Reduce Cost of GVT Calculation

The changes to the original method shown above are limited to the `while` loop. In this snippet of sample code it is clear that the number of events being evaluated has been significantly decreased through the elimination of the entire inner `for` loop.

```
/* This while loop iterates through all of the
processes, and looks for the advanceRequest message
with the lowest time. It then sets the nextGVT to that
time.  Note that this algorithm only checks each
process once for an advance request rather than
iterate through each processes entire queue.  */
while ( ((processIndex < numberOfProcesses) && (stop
== NULL)) ; processIndex ++)
```

```
{
     /* Checks the process for an advance request */
     if (e[processIndex]->advanceRequest)
     {
          /* sets the nextGVT to the advance request
          time if it is less than the current
          minimum*/
          nextGVT = min(nextGVT,
          e[processIndex]->advanceRequest);
     }
     /* Sends a null message if the queue is empty */
     else
     {
          /* stop calc*/
          stop = 1;

          /* Sends a NULL message.  Null messages are
          required to keep the process from entering a
          deadlock state. */
          sendNullMessage ( );

     } /* end if */

} /* end while */

if (stop == NULL)
{
     /* Sets the Global Virtual Time "GVT" to the new
     time obtained from the advance request */
     GVT = nextGVT;

     /* Publish Advance Grant message.  Once this is
     published processes are allowed to move to the
     requested time and execute events for those
     times.*/
     sendAdvanceGrant(GVT);

} /* end if */
```

This modification significantly improved the performance of calculating GVT. However, it did not improve the situation in which the slowest process was regulating the entire family of processes involved in CMB.

Lower Bound Time Stamp and Lookahead

The next major modification to CMB is often referred to as the Lower Bound Time Stamp (LBTS) method [Mattern93]. This takes advantage of the fact that most

simulations and games have a defined, discrete time step size. Under the traditional GVT method, when one process is operating on events at time 100, other remote processes are allowed to process all events up to and including those with stamp 100. However, using LBTS, remote processes recognize that a simulation operating at 100 right now will generate future event messages with timestamps of 100 plus one time step. Therefore, if a process' step size is 4, a remote simulation can be given permission to execute all events up to and including those with stamps of 104, knowing that the first process will not generate a message at 101 because it is not capable of it. Under the exact same conditions, LBTS is more aggressive than GVT and allows faster processes to move ahead of slower ones by some fraction of one step size. This can be extremely useful when different processes use different step sizes. In some cases, there are simulations with a step size of 1 working together with others using a step size of 2 or 3. (These are conceptual numbers that illustrate the ratio of size. An actual simulation process would use a step size such as 100 milliseconds, 200 milliseconds, or 1 second – which have ratios 1:2:10 and can be exploited by LBTS.) When this happens, the simulation using step size of 1 will find useful work to do with stamps or 103, when the simulation with step size of 2 would otherwise have regulated it back to 102 under GVT (Figure 3).

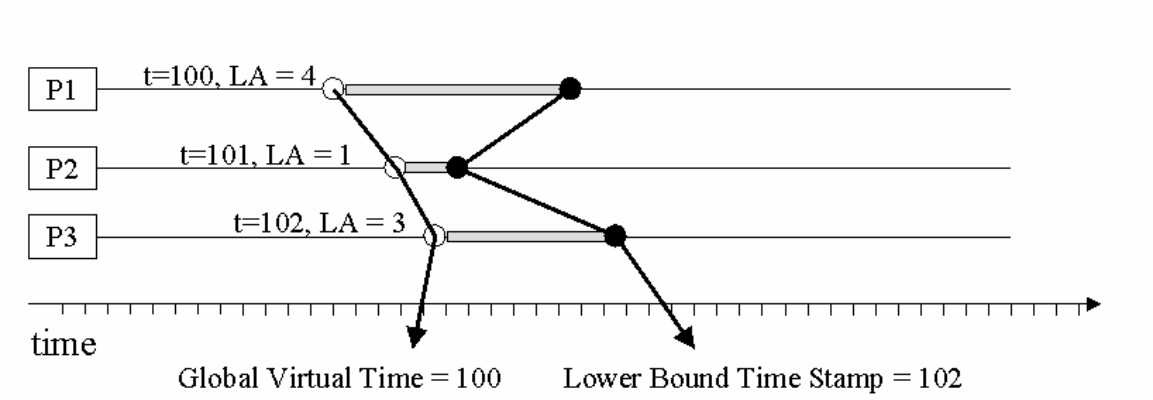

Figure 3. Comparison of GTV and LBTS

This slight modification can maintain causal consistency across the family of servers while also reducing the drag caused by the slower machines. The changes necessary to implement this are entirely limited to the `if` statement on `advanceRequest`. GVT is no longer equal to the lowest advance request value, but to the lowest of the sum of each processses' advance request and its lookahead value.

```
if (e[processIndex]->advanceRequest)
{
   nextGVT = min(nextGVT,
     (e[processIndex]->advanceRequest +
     e[processIndex]->lookAhead));
}
```

Deconflicting Simultaneous Timestamps

The discussion above provides the general solution to the synchronization problem. But there are several unique issues that must be dealt with to allow this mechanism to work. In a multiserver game, it is desirable to have all events processed in the same order on each server. This goes beyond time ordering them. It includes processing those with the same timestamps in the same order on multiple machines. Several interesting mechanisms have been created to support this [Fujimoto00].

Assume that the game time step size is 100 milliseconds. Then a timestamp value would have one hundred milliseconds as their smallest digit. A timestamp value of "12345" would represent 1,234 seconds and 500 milliseconds into the game. To support absolute ordering of events, this number must be augmented with more information. Some researchers point out that this is almost equivalent to stamping events with a unit smaller than the game's native step size. But, there other techniques that are a little more complex than that.

The first option is to add an ID number to the timestamp that is an incrementing counter that resets to zero each time the game ticks to the next step. This means that every event is tagged with a time, such as 12345, but also receives a counter. Therefore its time and ID number might be 12345.002, followed by an event stamped 12345.003, 12345.004, etc. This inclusion of an ID number works very well for indicating the order in which events are created. It also allows a receiving process to use this information to identify any event messages that are missing. In this paper we use '.' to delimit the information, which is a useful method for explaining the concepts. But in practice these values may fit into different digit positions within a single large integer or may be stored in different variables.

However, the order in which messages are created is not necessarily the order in which they should be executed. This has led to the practice of creating an "age" and a "priority" for messages. The "age" identifier is like a generational indicator. Events that are stored in the initial starting data set have an age of 0. When one of those events causes another event to be created, the created event has an age of 1. When that one causes an event, it will have an age of 2. If an event of age 3 triggers the creation of two new events, then both of them have an age of 4. This insures that whenever an event is caused by another event, the causal order between the two is maintained. When age is combined with the unique ID described earlier, it insures that an ordering algorithm always places two sibling events in the order that they were created. Adding age as part of the timestamp can be done in many ways, but we will illustrate it as "timestamp.age.ID" or "12345.002.004" in which 12345 is the actual timestamp, 002 is the age, and 004 is the unique ID.

Priority indicates specific events that should be processed ahead of others. These are usually events that have a need to be executed very quickly after being sent. For example, in an FPS game, any explosion events should have a higher priority than player-to-player chat messages. The use of a priority stamp can replace the use of age or be combined with it. Retaining all of these pieces of data may result in a timestamp that includes "timestamp.priority.age.ID".

The age and ID modifications were created to improve synchronization within a time step. Priority allows the sender to specify which events should be addressed first.

Conclusion

This chapter has described algorithms that are designed to synchronize event execution and time advance in distributed simulations and virtual worlds. These techniques have been in use for many years in the high performance computing community. As MMPGs grow more complex and comprehensive, they develop a similar need for strict synchronization between some or all of the servers within the server farm. The Chandy/Misra/Bryant algorithm and the modifications shown here are the most applicable event and time synchronization algorithms that can be applied in this environment. At one time the speed of computers and networks limited the use of these algorithms to simulations with timestep sizes on the order of 1 minute. However, as hardware performance has improved, it has been possible to bring these algorithms into simulations operating with 1 second or 100 millisecond timesteps. Continually improving hardware performance, algorithm optimization, and the complexity of distributed virtual worlds will make these methods accessible to simulations with even smaller timesteps in the future.

Like all new additions to game software, the computational costs and impacts of the new algorithm must be balanced against the benefits provided. Many games and even military simulations operate sufficiently well without strict synchronization. The purpose of MMPGs is to provide a believable immersive experience. The demands for accuracy are not as high as those for modeling nuclear blasts or chemical reactions. Algorithms like CMB will earn their way into MMPGs as they become better understood, the costs for implementing them are known, specific MMPG optimizations emerge, and the complexity of MMPG worlds increases.

References

[Beardsley03] Beardsley, Jason, “Seamless Servers: The Case For and Against”, *Massively Multiplayer Game Development*, Charles River Media, 2003.

[Fujimoto00] Fujimoto, Richard, *Parallel and Distributed Simulation Systems*. Wiley-Interscience, 2000.

[Fujimoto90] Fujimoto, Richard, “Parallel Discrete Event Simulation”. *Communications of the ACM*, Association of Computing Machinery, 1990.

[Johnson04] Johnson, Jeff, “Massively-Multiplayer Engineering”, *Proceedings of the Game Developers Conference*, available online at http://www.gdconf.com/, 2004.

[Mattern93] Mattern, Friedeman, “Efficient Algorithms for Distributed Snapshots and Global Virtual Time Approximation”, *Journal of Parallel and Distributed Computing*. Vol 18, Num 4., 1993.

[Perumalla04] Perumalla, Kalyan, "libSynk: Source Code for Time Synchronization", available online at http://www.cc.gatech.edu/computing/pads/kalyan/libsynk.htm, July, 2004.

[Smith00] Smith, Roger, "Synchronizing Distributed Virtual Worlds", available online at http://www.modelbenders.com/Bookshop/techpapers.html, December 2000.