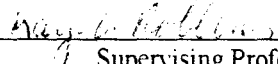

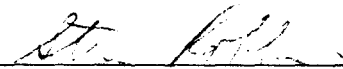



**Reliable Dynamic Terrain Updates and
Fault Tolerant Terrain Servers for
Distributed Interactive Simulations**

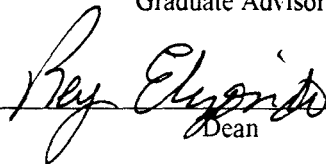
APPROVED: 
Supervising Professor



Hugh B Maynard



RECOMMENDED FOR ACCEPTANCE: 
Graduate Advisor

ACCEPTED: 
Dean

Reliable Dynamic Terrain Updates and
Fault Tolerant Terrain Servers for
Distributed Interactive Simulations

by

Richard M. Rybacki, No B.S.

THESIS

Presented to the Graduate Faculty of
The University of Texas at San Antonio
in Partial Fulfillment
of the Requirements
for the Degree of

MASTER OF SCIENCE

THE UNIVERSITY OF TEXAS AT SAN ANTONIO
May 1995

Acknowledgments

I would like to express appreciation to my advising professor, Dr. Kay Robbins, for her time and guidance throughout the evolution of this thesis, and for her contributions to my academic and professional development in general. I would also like to acknowledge TASC, Inc. for the monetary and material support which helped make this thesis possible.

Richard M. Rybacki

The University of Texas at San Antonio

Mail 1995

Contents

1 INTRODUCTION	6
1.1 DISTRIBUTED INTERACTIVE SIMULATION DEFINED	7
1.2 MOTIVATION FOR DYNAMIC TERRAIN	
2 THE DYNAMIC TERRAIN UPDATE PROBLEM	15
2.1 TERRAIN REPRESENTATIONS	15
2.2 THE RELIABLE CONNECTION-ORIENTED SERVICE APPROACH	17
2.3 THE HOST CHECK-IN APPROACH	18
2.4 THE PIGGY-BACKED ACKNOWLEDGMENT APPROACH	21
2.5 THE SEQUENCED UPDATE MESSAGE APPROACH	23
2.6 CURRENT RESEARCH ON RELIABLE BROADCAST	25
3 THE CHECKSUM TREE PROTOCOL	27
3.1 A DETAILED EXAMPLE	31
4 THE FAULT TOLERANT DYNAMIC TERRAIN SERVER PROBLEM	35
4.1 THE VIRTUAL SERVER ABSTRACTION	35
4.2 CONSISTENCY AMONG SERVERS	38
4.3 LEVEL I FAULT TOLERANCE	40
4.4 LEVEL II FAULT TOLERANCE	41
5 DETAILS OF IMPLEMENTATION.	43
5.1 THE TERRAINDB CLASS	43
5.2 THE CHECKSUM TREE CLASS	45
5.3 DIS PROTOCOL EXTENSIONS	46
5.4 CLIENT-SIDE PROCEDURAL CODE	48
5.5 SERVER-SIDE PROCEDURAL CODE	51
5.6 VIRTUAL SERVER IMPLEMENTATION	54
6 TESTING THE IMPLEMENTATION	61
6.1 THE STEALTH SIMULATOR	61
6.2 THE TRUCK SIMULATOR	62
6.3 THE DYNAMIC TERRAIN SERVER	64
6.4 THE PDU GENERATION TOOL	64
6.5 CHECKSUM TREE TESTING SCENARIOS	65
6.6 VIRTUAL SERVER TESTING SCENARIO	68
7 CONCLUSIONS AND SUGGESTIONS FOR FURTHER RESEARCH	70
7.1 CHECKSUM TREE TECHNICAL CONSIDERATIONS	71
7.2 VIRTUAL SERVER TECHNICAL CONSIDERATIONS	72
BIBLIOGRAPHY	74
VITA	77

1 Introduction

Aircraft simulators have been in use for many years for training pilots. Such a simulator provides the trainee with a mockup of the "cockpit" of the actual vehicle being simulated.¹ The out-the-window scene is typically produced by a computer image generator. The simulator allows the trainee to become familiar with the controls and displays of the actual vehicle, while providing a safer and less expensive alternative to training in the actual vehicle.

The Department of Defense uses simulators for advanced weapons training. The vehicles used in today's modern warfare are equipped with many complex weapon systems. Simulation allows troops to safely sharpen their skills on these complex systems in a simulation environment, while saving considerable expense.²

Simulators have been shown to be effective for training an individual in specific skills, however due to their standalone nature, they have not been a factor in collective, combined arms, joint training.³ In the past, the United States military has relied solely on field exercises to train individuals to perform as an integrated, coordinated unit. In recent years budget cuts, loss of land, and a heightened concern for environmental impact has lead the DoD to consider other alternatives to field training.⁴

1.1 Distributed Interactive Simulation Defined

Distributed Interactive Simulation (DIS) was conceived in the mid 1980s as a result of the ARPA SIMNET program. One of the goals of the SIMNET program was to define a networking protocol which would enable disparate simulators to interoperate in a distributed virtual reality environment. The original SIMNET protocol has evolved into the DIS protocol, which is now an IEEE standard.⁵ This standard defines a set of protocol data units (PDUs) through which entities participating in a distributed interactive simulation may exchange state information and communicate simulation events such as weapons fire, warhead detonation, or collisions with other simulators. The messages broadcasted onto the network communicate “ground truth”. The individual simulators which receive these messages are responsible for determining what is perceived.

The most frequently used PDU is the EntityStatePDU, which is used by entities to broadcast their state onto a network. EntityStatePDUs contain all the necessary information for a given simulator to render and interoperate with remote simulators.

Figure 1.1

```
struct EntityStatePDU
{
    PDUHeader          header;
    EntityID           entityID;
    unsigned char      forceID;
    unsigned char      numArticulatedParts;
    EntityType         entityType;
    EntityType         guise;
    LinearVelocity     velocity;
    WorldCoordinates   location;
    EntityOrientation  orientation;
    unsigned long      appearance;
    DeadReckonParms   deadReckonParms;
    EntityMarking      marking;
    EntityCapabilities capabilities;
    ArticulatedParts  articulatedParts[1];
};
```

The EntityStatePDU as defined by DIS Version 2.0 Draft 3 is illustrated in Figure 1. 1. This PDU contains a three-tuple identifier which uniquely identifies the vehicle that the PDU is from. Also contained in this PDU is the type of entity (i.e. M 1 tank, AH64 Apache helicopter, etc.), the location of the entity given in a three-dimensional earth-centered earthfixed coordinate system, velocity vector, orientation, and other miscellaneous information.

The virtual world created by a distributed interactive simulation takes place over some rectangular region of terrain which can be up to hundreds of kilometers across in any direction. This terrain surface is generally a polygon mesh created from Defense Mapping Agency elevation data. The terrain is typically augmented with features as well. These features include trees, tree lines, forested areas, roads, rivers, railroads, lakes, buildings, utility poles and lines. The terrain and features which comprise the virtual world are persistent and static in the sense that they are not mutable by events of the simulation.

In contrast to the terrain and features, vehicles operating in the virtual world are quite dynamic. They may assume any location, orientation, or appearance within the constraints of the vehicle being simulated. Crews operating a simulator view the virtual world from the vantage point of their vehicle. Because the simulators broadcast their state information (i.e. location, orientation, velocity) over a network, the out-the-window scene produced by simulators may include not only terrain and features but also vehicles generated by other remote simulators.

This technological breakthrough allowed large multforce virtual battles to be created by networking simulators for tanks, armored personnel carriers, rotary wing and fixed wing

aircraft. Unlike the case of the standalone simulator in which the trainee only learns to operate the simulated vehicle, Distributed Interactive Simulation allows troops to train together to learn tactics and doctrine, command and control, and communications as well.⁶

Since its inception, DIS has broadened in scope to include virtual prototyping, systems testing, human factors analysis, and the evaluation of threat systems. DIS today is often used to influence engineering decisions concerning proposed weapon systems, and also to develop tactics and doctrine for existing systems which are not yet battle proven.⁷ The potential of DIS is being recognized by non-military organizations as well. DIS is likely to play a major role in the evolution of the Intelligent Vehicle Highway System.⁸ Many state governments are investigating the feasibility of DIS for emergency response training.⁹ DIS is being investigated as a training technique for air traffic controllers,¹⁰ driver's training,¹¹ and it is not difficult to imagine the potential of DIS in the entertainment industry.

1.2 Motivation for Dynamic Terrain

One of the major drawbacks of the DIS architecture is that the terrain is static over the course of a simulation. A more realistic simulation would allow for simulation events to effect the terrain surface and features. A large bomb exploding over the terrain surface is likely to create a crater, making that section of the terrain impassable for many types of vehicles. A bulldozer should be able to create a bank behind which tanks may take cover in a defensive position. An aircraft should not be able to take off from a runway which has been cratered by bombing.

In static terrain simulations, all entities are assured to have a consistent view of the virtual environment because terrain and associated features are loaded from disk files at simulation initialization time. A dynamic terrain simulation however would require updates to be transmitted over the network. To date, dynamic terrain has been considered by the DIS community to be economically unfeasible. This is due in part to the fact that terrain and feature data are used in a variety of different ways, and often a different representation is warranted by the different uses. For example, image generation systems producing the out-the-window scene require the terrain to be represented by a database of polygons with associated color and texture information.¹² Image generators account for the majority of the cost of a simulator, and terrain database formats used by image generators are proprietary, highly coupled to the particular image generation system, and not readily amenable to real-time updates.¹³ A polygonal representation of the terrain is also not suitable for an automated forces workstation which injects unmanned vehicles into the virtual world and hence does not require an image generator.¹⁴ This type of application requires a terrain database format more suitable for tasks such as calculating the elevation of the terrain at a given X-Y coordinate, orienting a ground vehicle onto the terrain surface given an X-Y location and heading, or determining the existence of a line of sight between two points.

Terrain representation schemes which are amenable to dynamic updates and also meet the needs described above are the topic of many papers being presented at the DIS workshops.¹⁵ However, the implications of dynamic terrain on the DIS protocol has been largely ignored.¹⁶ Due to the persistent nature of terrain, it is necessary that all simulation

participants have a consistent view of the terrain database. If simulation events are allowed to affect the terrain surface, it has been the consensus of the DIS community that the responsibility for mutating the terrain should be accomplished by a dynamic terrain server.¹⁷ The dynamic terrain server would monitor the network traffic looking for messages which would be capable of mutating the terrain. Examples of such messages would include DetonationPDUs, which indicate the location and warhead type of a detonation, and CollisionPDUs which indicate collisions between two vehicles or a collision between a vehicle and the terrain. Like a static terrain simulation, a dynamic terrain simulation would require the initial state of the terrain database to be loaded from disk files by simulation participants at initialization time. Throughout the course of a simulation, changes to the terrain affected by the dynamic terrain server must be communicated to the simulation participants through the use of PDUs added to the DIS protocol.¹⁸

Currently, all DIS PDUs are broadcasted onto the network using UDP, an unreliable datagram service.¹⁹ Due to the large number of entities which may participate in a distributed simulation, reliable point-to-point communication of state messages would use an extensive amount of network bandwidth and would not lend itself well to the real-time nature of a distributed interactive simulation.²⁰ EntityStatePDUs, which represent the vast majority of network traffic, fall into this category.²¹ If a given simulator A were to miss an EntityStatePDU from another simulator B, A would be able to dead reckon the position of B using the position and velocity vector provided by the last EntityStatePDU received from vehicle B.²² If the EntityStatePDU missed by vehicle A communicated a significant change

in the velocity vector, then the dead reckoned position of vehicle B would vary significantly from the "truth" position of vehicle B. At the point in time when vehicle A receives the next EntityStatePDU from vehicle B, vehicle B would appear to jump from its dead reckoned position to its truth position from the vantage point of vehicle A. The degree to which this momentary visual anomaly manifests itself is a function of the change in velocity vectors and the elapsed time between the EntityStatePDU missed by simulator A, and the next EntityStatePDU which A receives from B. At the point which simulator A finally receives the EntityStatePDU from simulator B, the situation has corrected itself with no cumulative detriment to the simulation.

Unlike the case of the EntityStatePDUs, PDUs which contain updates to the terrain database must be transmitted in a reliable fashion. If the dynamic terrain server mutates a section of the terrain and one or more of the simulation hosts does not receive the update, the validity of the simulation is in jeopardy. In the case of manned simulators, human operators make decisions based on visual feedback from scenes rendered by image generation systems. If they are presented with a scene which is not consistent with the current state of the virtual environment, it is unlikely that they will be able to take the appropriate action. This is equally true in the case of automated forces, which use their notion of the environment to perform terrain reasoning. For both manned simulators and automated forces, accurate terrain information is necessary for orienting the vehicle in space and calculating vehicle dynamics.

Due to the persistent nature of terrain, missed updates to the terrain database would cause visual anomalies which would remain for the duration of the simulation. As an example, consider a tank simulator which misses an update to the terrain database which results in a crater in the pathway of the tank. As the tank simulator passed over the newly formed crater, the simulation software would be using out-of-date terrain information for orienting the vehicle in space, calculating vehicle dynamics, and rendering the out-the-window scene. From the vantage point of other vehicles, the tank would appear to be flying over the crater. The crew members in the tank would not be able to see the crater either. If the tank simulator would have received the terrain update and drove into the crater, this may have resulted in a thrown track for the tank. Clearly, missed terrain update messages could effect the outcome of a simulation.

In order to maintain consistency with the DIS philosophy of no reliance on a central computer, multiple redundant dynamic terrain servers must be able to coexist on the same simulation network. However, there should not be multiple responses to simulation events which mutate the terrain surface. If a particular server goes off line, this should be detected by the other server(s) and some server should take over as appropriate.

As one can infer from the above discussion, the implementation of dynamic terrain in distributed interactive simulation imposes many interesting network protocol related challenges. Chapter 2 discusses some potential solutions to the problem of providing reliable terrain updates, and shows why these solutions would not be feasible. Chapter 3 then introduces the Checksum Tree, which is the solution chosen by this thesis to address the

problem of reliable terrain updates. The Checksum Tree is a novel approach to representing the state of the entire dynamic terrain database. The terrain database state is then broadcasted onto the network by the terrain server so that client simulations may confirm that their local copy of the terrain database is up-to-date. Chapter 4 addresses the issue of fault tolerance, and introduces the concept of the Virtual Server for providing a fault tolerant dynamic terrain serving entity. The Virtual Server is also a new idea for addressing the need for fault tolerance among environmental servers. Chapter 5 discusses the details of implementation of the Checksum Tree and the Virtual Server. Chapter 6 discusses the methods and results of testing the implementation. Chapter 7 provides conclusions and suggests areas for further research.

2 The Dynamic Terrain Update Problem

The introductory chapter established the need for reliable transmission of terrain update messages. This chapter begins by introducing terrain representations used for distributed interactive simulations. The shortcomings to some of the obvious approaches to the dynamic terrain update problem are discussed, followed by an examination of the existing literature. The following chapter then introduces the Checksum Tree Protocol, which is the solution purposed by this thesis.

2.1 Terrain Representations

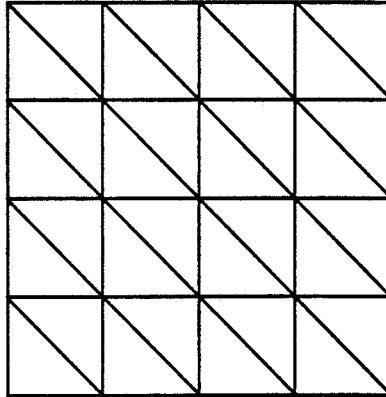
Terrain databases are typically on the order of megabytes in size. The database must be segmented into manageable size units which are suitable for network transmission. In this discussion we will refer to these units as terrain "patches". The driving factors in choosing the size of the patch are the maximum possible size of the terrain update message and the desired maximum fidelity or resolution in which the terrain is represented. The DIS protocol specifies the User Datagram Protocol (UDP) as the underlying transport mechanism. When using Ethernet V2.0 as the underlying data link service, the maximum size of a DIS PDU is 1458 bytes.²³

Most systems used for DIS today use a polygonal mesh to represent the terrain surface. Alternate terrain representations which better lend themselves to network transmission are currently being studied. Examples of such representations include B-Splines and parametric surfaces.²⁴ The emphasis of this thesis however is not on terrain representation but on the handshaking mechanism between terrain servers and simulation participants to provide reliable data transfer.

For the sake of simplicity, the terrain representation used by this thesis ignores any terrain features and represents the terrain surface as a 2-dimensional grid of elevation samples. From this elevation grid, a polygon mesh²⁵ may be formed as illustrated by Figure 2.1. Four mutually adjacent elevation samples form the vertices

for two triangles by placing an edge between each elevation sample and its immediate South-East neighbor. Edges are also placed between horizontally and vertically adjacent elevation samples.

Figure 2.1



A patch size of 500m by 500m using 32-bit floating point numbers to represent the elevation data allows for an elevation sample every 26.3 meters. A patch size of 100m allows for an elevation sample every 5.3 meters. In the interest of optimal usage of the network bandwidth, tradeoffs must be made between the patch size and the maximum resolution of the terrain. For the purposes of illustrating the concepts presented in this thesis, a patch size of 500m by 500m was chosen.

Associated with each patch of terrain is a revision level. Initially the revision level of all patches is zero and is incremented each time the patch is modified and transmitted by the dynamic terrain server. The following sections discuss possibilities for ensuring the reliable transmission of terrain update messages.

2.2 The Reliable Connection-Oriented Service Approach

To perform reliable data transfer over a network, one typically chooses a protocol in which the data packets to be transferred from the source host to the destination host are acknowledged by the receiving host. The Transmission Control Protocol (TCP)²⁶ is an example of a reliable connection-oriented service which employs such a mechanism to provide reliable data transfer. When trying to layer such a technique on top of the DIS protocol however, some difficulties arise:

- DIS, being layered above UDP, is an unreliable broadcast service. If the terrain server were to establish a reliable connection-oriented service to each participant in the simulation, a large amount of redundant data would be transmitted over the network.
- DIS simulations can involve up to thousands of entities.²⁷ As the terrain updates must take place in real time, it is not feasible for the terrain server to establish a reliable connection-oriented service to each simulation participant, for the delays would far exceed the human perception limit of 100 milliseconds.²⁸

A reliable connection-oriented service is not a feasible option considering the current network technology due to the amount of bandwidth which would be consumed by the point-to-point connections established between a dynamic terrain server and numerous simulation clients. Furthermore, the temporal latencies associated with a server individually servicing potentially hundreds of clients would not be tolerable in a real-time simulation.

2.3 The Host Check-In Approach

The Host Check-In (HCI) approach is similar to a reliable point-to-point connection-oriented service, with the exception that the terrain update data itself need not be transmitted redundantly over the network to multiple simulation participants. The HCI approach would require each simulation host to check in to a particular terrain server at initialization time. Simulation participants checked-in to a particular server become clients of that server. The process of checking in informs the server of the number and identity of each of its clients. Reliable transfer of terrain updates is afforded by the fact that each server expects to receive an acknowledgment packet from each of its clients when a terrain update message is transmitted. At the expiration of some time-out period, the server can determine which clients failed to acknowledge the terrain update message. If some clients fail to acknowledge, the server will retransmit the message. This process may continue for some number of iterations, at which point non-responding clients may be assumed by the server to be off line.

When the terrain server updates a patch of terrain in the database, it issues a single TerrainPatchUpdatePDU. The terrain server then expects to receive a TerrainPatchUpdateAckPDU from each of its clients. Figure 2.2 illustrates the TerrainPatchUpdatePDU and the TerrainPatchUpdateAckPDU. At the expiration of some time-out period, the server is able to determine if any of its clients fail to acknowledge a TerrainPatchUpdatePDU. If this is the case, another TerrainPatchUpdatePDU will be issued. Note that the TerrainPatchUpdatePDU contains a

Figure 2.2

```
struct TerrainPatchUpdatePDU
{
    PDUHeader          header;
    SimulationAddress  serverID;
    unsigned short     terrainDatabaseID;
    unsigned short     revisionNumber;
    unsigned short     patchCoordX;
    unsigned short     patchCoordY;
    unsigned short     sampleSize;
    short              _unused;
    float              zValues[1];    // Varies
};

struct TerrainPatchUpdateAckPDU
{
    PDUHeader          header;
    SimulationAddress  clientID;
    SimulationAddress  serverID;
    unsigned short     terrainDatabaseID;
    unsigned short     revisionNumber;
    unsigned short     patchCoordX;
    unsigned short     patchCoordY;
};
```

revision level for the patch which is being updated. When a simulation receives an update for a patch of terrain it checks the revision level of that patch in its local copy of the terrain database. If the revision level is the same as that which is specified by the PDU, the update is ignored. For this reason, repeated broadcasts are idempotent with respect to clients.

However, an acknowledgment is still required from every client as the original acknowledgment may have been lost.

The best case scenario (i.e. no lost packets or acknowledgments) using the HCI approach would result in $N+1$ packets being transmitted for each terrain patch update, where N is the number of client simulations. In comparison with the Reliable Connection Oriented Service where $2*N$ represents the best case scenario, HCI is definitely a preferred approach. Simulation hosts also benefit from receiving their patch update messages simultaneously during the initial broadcast, as opposed to waiting in turn for the server to establish a connection with each client.

The HCI approach is promising in scenarios where the number of simulation hosts is small. Note that this does not limit the number of *entities* which participate in a simulation, since one host may simulate numerous entities. Simulations involving a large number of entities which are generated primarily by automated forces workstations may best benefit from the HCI approach. However, simulations which involve large numbers of manned entities, where the simulation host to entity ratio approaches one, may find that the number of acknowledgments generated for each terrain patch update becomes a significant impact on network bandwidth.

A significant disadvantage to the HCI approach arises when either a client fails to receive the TerrainPatchUpdatePDU, or the corresponding TerrainPatchUpdateAckPDU is lost. If either of these happen, the server does not receive acknowledgments from all clients and transmits another TerrainPatchUpdatePDU. However, the server has no way of knowing

if a client failed to receive the TerrainPatchUpdatePDU or if the acknowledgment was lost. This forces all clients to again transmit acknowledgments even though they may have originally received the TerrainPatchUpdatePDU, and their acknowledgment was received by the server. Thus each retransmittal by the server results in another N+ I packets on the network. During conditions where a high probability exists for collisions, this condition could lead to an unstable situation.²⁹

2.4 The Piggy-Backed Acknowledgment Approach

Throughout the course of a simulation, simulators are emitting EntityStatePDUs at the rate of at least the inverse of the time-out threshold specified by the exercise administrator. The DIS standard dictates five seconds for the time-out threshold. As entities move about the virtual world, their dead-reckoned positions and orientations begin to deviate from their "truth" positions and orientations eliciting the emission of even more EntityStatePDUs. The Piggy-Backed Acknowledgment (PBA) approach views the EntityStatePDU traffic as a "sunk cost" and attempts to piggyback acknowledgments to dynamic terrain update messages onto these EntityStatePDUs. This approach would expand the EntityStatePDU to include the patch number and revision level of the patch the entity is currently located over. Figure 2.3 illustrates the modified EntityStatePDU, with the inserted fields indicated in bold.

Terrain servers monitor EntityStatePDUs to ensure that revision levels are up to date for the specified patch. If an entity is using an incorrect revision level of the patch, the server can infer that the particular entity did not receive the original patch update message, and a new patch update message will be issued by the server.

Note that the patch is identified by its normalized coordinates, patchCoordX and patchCoordY, which are relative to the origin of the terrain database. These patch coordinates could be derived from the reported location of the vehicle in geocentric coordinates, however this would impose significant computational complexity on the servers who must check each and every EntityStatePDU on the network.

This technique lends itself well to the use of multiple terrain servers, where each server is responsible for a particular geographic partition of the terrain database. Under this arrangement, a server need only check EntityStatePDUs from vehicles which are operating within their geographic partition.

Figure 2.3

```
struct EntityStatePDU
{
    PDUHeader          header;
    EntityID           entityID;
    unsigned char      forceID;
    unsigned char      numArticulatedParts;
    EntityType         entityType;
    EntityType         guise;
    LinearVelocity     velocity;
    WorldCoordinates   location;
    EntityOrientation  orientation;
    unsigned long      appearance;
    DeadReckonParms   deadReckonParms;
    EntityMarking      marking;
    EntityCapabilities capabilities;
    unsigned short    patchCoordX;
    unsigned short    patchCoordY;
    unsigned short    revisionLevel;
    short             _unused;
    ArticulatedParts   articulatedParts[1];
};
```

PBA has the obvious advantage over both the Reliable Connection-Oriented Service and Host Check-In in that no overhead PDUs are generated. In the best case scenario, the only impact on network traffic is the TerrainPatchUpdatePDU itself. One subtle disadvantage is that simulators may use terrain patches other than the one they are currently positioned over for the purposes of out the window scene generation, clear line of sight calculations, terrain reasoning, etc. If a simulator were to miss a TerrainPatchUpdatePDU, and the affected patch of terrain were used by the simulator for rendering an out-the-window

Figure 2.4

```
struct TerrainPatchUpdatePDU
{
    PDUHeader          header;
    SimulationAddress  serverID;
    unsigned short     terrainDatabaseID;
    unsigned short     revisionNumber;
    unsigned short     patchCoordX;
    unsigned short     patchCoordY;
    unsigned short     sampleSize;
    short              unused;
    unsigned long      sequence;
    float              zValues[1]; // Varies
};
```

scene, the server would have no knowledge of the fact that the simulator in question was using a stale patch of terrain.

2.5 The Sequenced Update Message Approach

This approach is similar to the Piggy-Backed Acknowledgement approach discussed in the previous section, but is based on an idea purposed by Kaashoek.³⁰ In this approach, the dynamic terrain server would sequence terrain update messages. The

Figure 2.5

```
Struct EntityStatePDU
{
    PDUHeader          header;
    EntityID           entityID;
    unsigned char      forceID;
    unsigned char      numArculatedParts;
    EntityType         entityType;
    Entitytype         guise;
    LinearVelocity     velocity;
    WorldCoordinates   location;
    EntityOrientation  orientation;
    unsigned long      appearance;
    DeadReckonParms   deadReckonParms;
    EntityMarking      marking;
    EntityCapabilities capabilities;
    unsigned long      highestSequence;
    ArticulatedParts  articulatedParts[1];
};
```

TerrainPatchUpdatePDU is expanded to include this global sequence number assigned by the server and incremented for each new TerrainPatchUpdatePDU generated. Figure 2.4 illustrates the TerrainPatchUpdatePDU with the modified fields indicated in bold.

The EntityStatePDU is modified to include the highest sequence number for which the entity has received TerrainPatchUpdatePDUs for all lower sequence numbers. For example, if the entity receives TerrainPatchUpdatePDUs with sequence numbers 1, 2, 3, and 5; the entity reports 3 as the highest contiguous sequence number received in EntityStatePDUs generated by that entity. Figure 2.5 illustrates the EntityStatePDU with the modified fields indicated in bold.

The server is capable of determining if any terrain update messages were lost by monitoring EntityStatePDUs and checking the *highestSequence* field reported by the entities. In the above example, the current sequence is 5 and the entity in question is reporting 3. The server can infer that the entity missed at least the TerrainPatchUpdatePDU with sequence number 4, and possibly 5 as well. The server can then retransmit the

TerrainPatchUpdatePDU with sequence 4. If the entity missed sequence 5 also, that will be determined from subsequent EntityStatePDUs generated by that entity.

Assuming the server has finite memory resources and cannot maintain an infinite history of all TerrainPatchUpdatePDUs generated, client simulators will be required to check into the server. The server need not maintain TerrainPatchUpdatePDUs with sequence numbers less than or equal to the minimum of sequence numbers reported by all clients in EntityStatePDUs, and may purge its buffer of all TerrainPatchUpdatePDUs with sequence numbers up to and including this minimum sequence number.

This approach is effective if all entities are present and checked into the server before any TerrainPatchUpdatePDUs are generated. However, this protocol by itself does not support entities which join the simulation late - possibly after the server has flushed part of its buffer of TerrainPatchUpdatePDUs issued in the past. This protocol also requires the server to maintain knowledge of all entities participating in the simulation. The set of entities participating in a distributed interactive simulation is a dynamic one. Entities often join simulations late or exit early. Any change in the set of entities participating in the simulation must be reliably communicated to the server for this protocol to work effectively.

2.6 Current Research on Reliable Broadcast

The Dynamic Terrain Update Problem can be generalized to the problem of reliable broadcast. Much research has been done in this area^{31,32,33} The protocols presented in the literature are more general in that they allow for the broadcast of arbitrary data. Many of the

protocols also allow for any host to issue broadcast messages. A common characteristic of all reliable broadcast protocols presented in the literature is the requirement of acknowledgment messages, either explicit or piggy-backed onto existing messages in the data stream.

Acknowledgments imply that the broadcasting host know the number and identity of each of the message recipients. The ISIS³⁴ protocol is an example of a reliable broadcast protocol in which group maintenance is important to the correct operation of the protocol. This is a complication in a real-time distributed interactive simulations where entities frequently join and exit the simulation.

The Dynamic Terrain Update Problem has some characteristics which are not present in the more general cases addressed in the literature. The first characteristic is that the structure of the data contained in the broadcast messages is known. Secondly, there is a single known host which is performing the broadcasts - the dynamic terrain server. Client simulators are simply receivers of the broadcast messages and have no need themselves to perform reliable broadcast. Finally, there is no need for a server to maintain TerrainPatchUpdatePDUs which have been overridden by newer TerrainPatchUpdatePDUs for the same patch of terrain. The server need only maintain the current state of the terrain database. It serves no purpose to provide an entity with a terrain patch which is already stale. The following chapter introduces a protocol which exploits these problem domainspecific characteristics to provide an efficient mechanism for the reliable broadcast terrain update messages.

3 The Checksum Tree Protocol

The techniques discussed in the previous chapter attempted to provide absolute reliability of terrain updates using either explicit acknowledgments (HCI) or implicit acknowledgments (PBA). The shortcomings of these techniques however, has been shown to be intolerable. If the revision levels of all patches in the entire terrain database could be represented in some efficient manner, this information could be broadcasted by the dynamic terrain server in periodic "heartbeat" messages. A client simulation could check these revision levels against its own version of the terrain database to determine if any terrain update messages have been missed. If a client simulation has in fact missed terrain update messages, it may request a retransmission from the terrain server. The difficulty in this approach is that there are on the order of thousands of patches in even the smallest of terrain databases, so such a message would not be suitable for transmission in broadcast datagrams.

The Checksum Tree is a novel approach to representing the state of the entire dynamic terrain database with a single 32 bit unsigned integer which contains the "checksum" for the database. Instead of relying upon acknowledgments from client simulations, the server will simply broadcast the checksum of the terrain database in periodic "heartbeat" messages at some predetermined interval. It will then become the responsibility of the client simulations to determine if they have missed any terrain update messages. Through a small number of queries and responses between a client and a server, terrain

patches which are not consistent between the client and server may be quickly identified. The server may then retransmit the TerrainPatchUpdatePDU which was missed by the client.

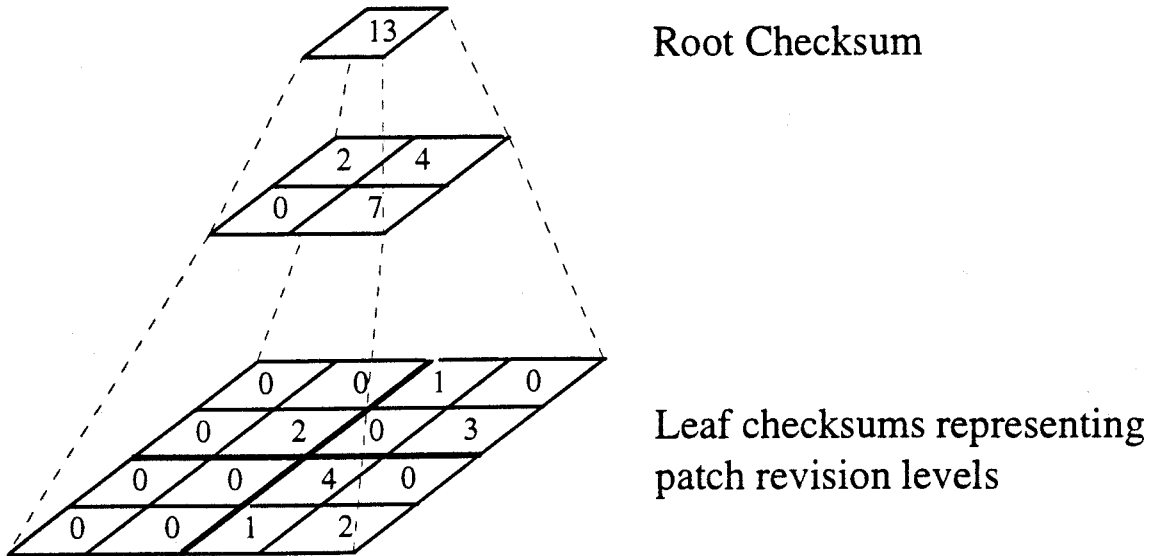
One underlying assumption in this approach is that missed terrain update messages by clients are relatively infrequent. Another underlying assumption is that some latency (on the order of tenths of a second) is tolerable between the time in which a client simulation becomes aware that its version of the terrain database is not consistent with that of the server, and the time in which the inconsistency is corrected.

In this approach, groups of patches are recursively aggregated into $N \times N$ nodes of a checksum tree. Associated with each node in the tree is a checksum. A leaf represents a patch of terrain and the associated checksum is the revision level of the patch. At the next higher level, a node consists of an $N \times N$ grid of patches. The checksum of such node is the sum of all the revision levels of each of the N^2 patches which comprise that node. This process continues recursively until the entire database is covered, at which point a single checksum represents the entire database. The checksum of the root node is actually the sum of all the revision levels of each patch in the entire terrain database. A 32-bit unsigned integer used to represent the checksum allows for up to $2^{32}-1$ terrain mutations before any ambiguity in the checksum can arise.

There are two factors to consider when choosing N . With a larger N , fewer iterations are required between the client and the server to resolve which terrain patch is stale from the overall checksum. In fact, the number of iterations required is:

$$\log_N(\text{number of patches in database})$$

Figure 3.1



Another factor to consider is the server response to a client checksum query must fit into a single UDP datagram. Choosing N to be 10 while using 32-bit checksums, results in 400 bytes of information which can easily be accommodated by a UDP datagram, including any necessary packet header information. Figure 3.1 demonstrates the checksum tree concept using 2x2 patch grids for illustration purposes.

Consider the Hunter-Ligget database, a 50Km by 50Km grid consisting of 100x100 500 meter patches. Assuming $N=10$, a three level checksum tree is required to cover the entire database. A four level checksum tree would cover a database as large as 500Km by 500Km.

Under the Checksum Tree Protocol, servers would transmit terrain patch updates in a broadcast manner at the point of terrain mutation as in HCI and PBA, but would also

broadcast periodic "heartbeat" messages at some determined interval. The heartbeat message would contain the checksum for the root node of the checksum tree, representing the entire database. When clients receive the heartbeat message, they may confirm whether their local copy of the terrain database is consistent with that of the server. If it is not, the resolution process is initiated. The following example illustrates how this may work where N is 10 and each node contains 100 (N^2) checksums:

1. A client disagrees with the checksum indicated by the heartbeat message. The client sends a query to the server asking for the 100 checksums which were used to calculate the checksum for the root node.
2. The server responds by issuing a packet containing the 100 checksums used to calculate the checksum for the root node.
3. The client determines that the checksum for child P of the root node is in disagreement with the server's checksum for that node. The client issues a query to the server for the 100 checksums which were used to calculate the checksum for child P of the root node.
4. The server responds by issuing a packet containing the 100 checksums used to calculate the checksum for child P of the root node.
5. The client determines that the checksum of child Q of node P is in disagreement with the server. The client sends a query to the server requesting the 100 checksums used to calculate the checksum for child Q of node P.
6. In the case of a three level Checksum Tree, we are already at a leaf node of the tree and the stale patch has been identified. The server responds by transmitting the TerrainPatchUpdatePDU for that patch.

The server issues repeated TerrainPatchUpdatePDUs in a broadcast manner so that if other clients missed the same original TerrainPatchUpdatePDU, they may benefit from the repeated broadcast as well. Note that repeated broadcasts of terrain update messages are idempotent because the message contains the revision level for the patch, and clients may ignore these messages if they are using the proper revision level for the patch.

When a client receives a TerrainPatchUpdatePDU, it recalculates the entire database checksum by assigning the revision level for the updated patch to the corresponding leaf node of the checksum tree. The checksum tree is then ascended originating at the affected leaf, accumulating sums at each level until the root node is reached, at which point the checksum for the entire database is known. Note that as a client visits each node in the ascent of the tree, resuming the entire node is not necessary. The client may simply add the difference between the new checksum and the previous checksum of the affected child to the checksum for that node.

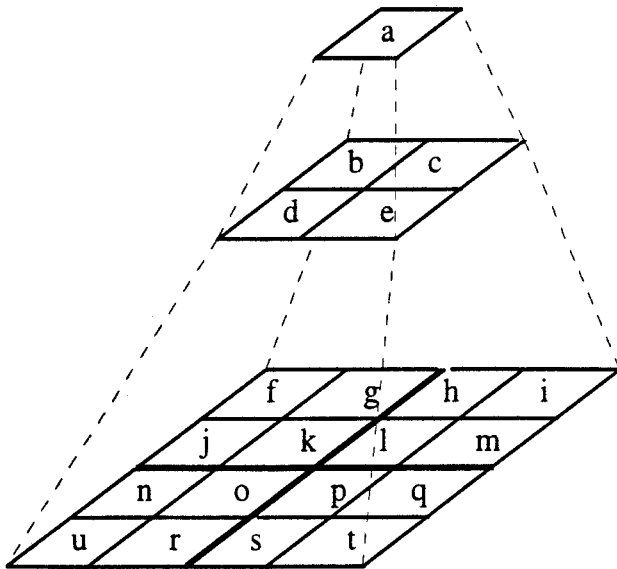
The dynamic terrain server itself also maintains a checksum tree data structure. Each time a TerrainPatchUpdatePDU is issued by the server, the revision level for that patch of terrain is incremented and assigned to the corresponding leaf node its checksum tree. As in the case of the client, the tree is ascended to compute the root checksum. The root checksum is then available for transmission in heartbeat messages.

3.1 A Detailed Example

The following provides a detailed example of how the Checksum Tree Protocol is used between a client and a server to resolve missed terrain update messages. In the scaled-down example, the terrain database consists of 16 patches. Patches are aggregated into 2x2 nodes forming a three level checksum tree as illustrated in Figure 3.2. For illustration purposes, we establish a convention for labeling the nodes within a checksum tree. *a* is the

label of the root node; b, c, d, e are the labels of a 's children, and so on. Figure 3.2 illustrates the node labeling convention.

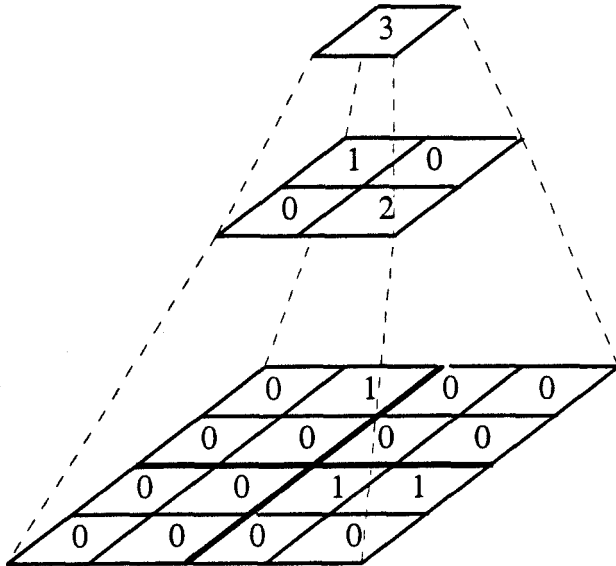
Figure 3.2



Consider the following scenario. A client's network connection is momentarily offline while the server transmits three terrain update messages. The checksum tree for the server at this point in time is illustrated in Figure 3.3.

At time t the server transmits a heartbeat message indicating that the root checksum is 3. The client, having missed all terrain update messages due to being off-line, has a root checksum of 0.

Figure 3.3



At time t_1 the client receives the heartbeat message. Since the root checksums disagree, the client issues a request for the checksums of node a . At time t_2 , the server receives this request and responds with $a = \{ 1, 0, 0, 2 \}$. Note that the checksums indicated by the response correspond to the checksum tree nodes $\{ b, c, d, e \}$. The client receives the response at time t_3 . It is determined by the client that the checksums for nodes $\{ b, e \}$ are in disagreement with the server. The client issues then issues two separate requests, one for node b and another for node e . At time t_4 and t_5 , the server receives these requests and responds with $b = \{ 0, 1, 0, 0 \}$, and $e = \{ 1, 1, 0, 0 \}$, respectively. Upon receipt of the server responses, the client will determine that nodes

$\{ g, p, q \}$ are in disagreement with the server and issues requests for these nodes. Since these are leaf nodes of the tree, the server responds by retransmitting the terrain patch update message for each of these patches.

As the client receives the terrain update messages for patches $\{ g, p, q \}$, the associated revision numbers for these patches are assigned to the corresponding leaf nodes of the client's checksum tree. If all messages are received, the client's checksum equals the server's checksum when the next heartbeat message is received from the server. Otherwise, the resolution process is reinitiated to determine which patches remain stale. If any of the Checksum Tree Protocol messages are lost on the network, the root checksums are still in disagreement on the receipt of the next heartbeat message from the server, and the resolution process is reinitiated.

4 The Fault Tolerant Dynamic Terrain Server Problem

The Checksum Tree Protocol discussed in the previous chapter was shown to be a promising approach to providing efficient and reliable terrain updates. The Checksum Tree by itself however does not address the issue of fault tolerance. Other techniques must be established in conjunction with the Checksum Tree to afford the simulation a reliable terrain serving entity. This thesis purposes a concept called the "*Virtual Server*" which attempts to address this need. Although a detailed and correct implementation of a distributed fault tolerant terrain server is beyond the scope of this thesis, the following discussion is provided to stimulate thought on this subject.

4.1 The Virtual Server Abstraction

In order to remain consistent with the DIS philosophy of no reliance on a central computer³⁵ the dynamic terrain server must actually consist of two or more cooperating servers to ensure fault tolerance.³⁶ The *virtual terrain server* is an abstraction employed by the client simulations to hide the fact that there may be multiple redundant servers coexisting on the network. The existence of multiple servers on the network, and the interaction between these servers is hidden from the client simulations. The client simulations receive terrain updates and heartbeat messages from, and issue queries to, the virtual server.

From a simulation client's point-of-view, a single, reliable, terrain serving entity exists on the network at all times.

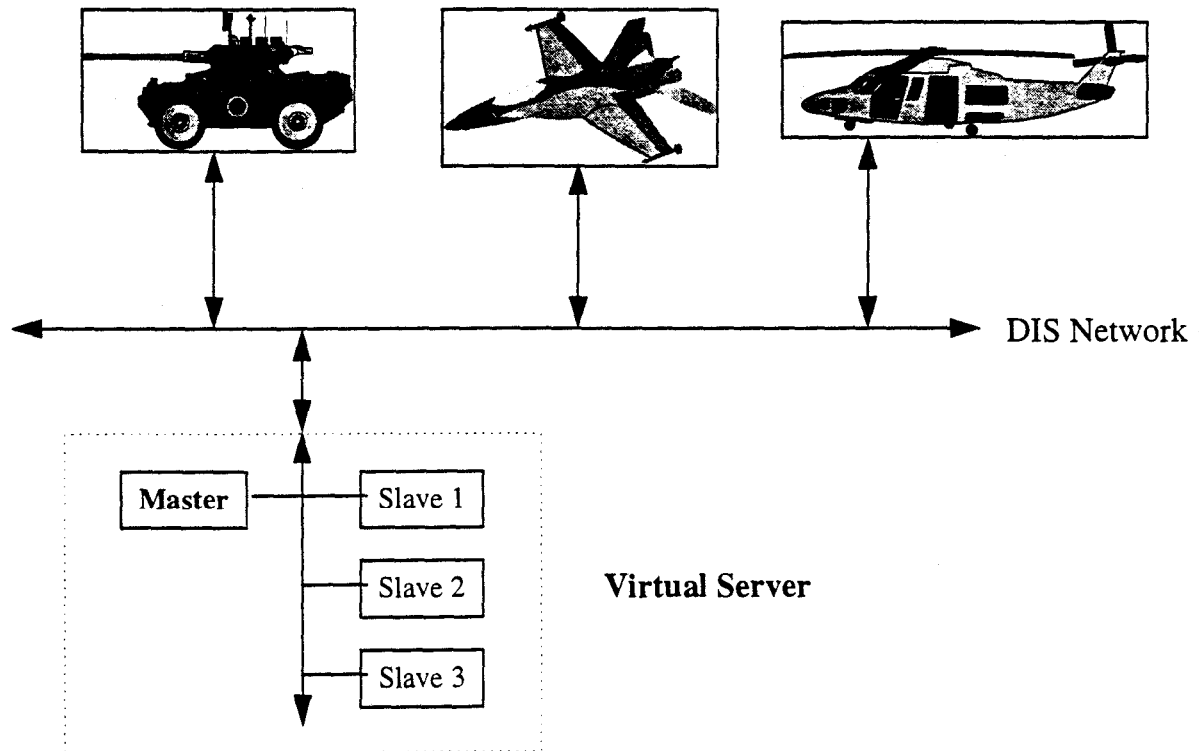
The abstraction of the *virtual server* is necessary to minimize the impact on clientside code. Advantages to this approach are both technical and political. From the technical perspective, the *virtual server* requires no client-side implementation of the protocol. This reduces likelihood of errors and provides a more elegant solution by encapsulating the virtual server complexity solely within the server code. From the political perspective, the concept will be more "sellable" to the DIS community because no special client-side modifications are necessary beyond the implementation of the Checksum Tree Protocol. Imposing the implementation of complicated protocol algorithms upon the simulation clients is likely to discourage participation in distributed simulations.

The virtual server provides periodic heartbeat messages so that clients may ensure that their view of the terrain is up-to-date. The virtual server broadcasts terrain update messages when the terrain is mutated by a simulation event. Clients may communicate with the virtual server for the purposes of checksum discrepancy resolution, and late joining clients query the virtual server so that they have an up-to-date view of the environment. Figure 4.1 illustrates the virtual server concept and its relationship to simulation clients.

It is undesirable for all of the servers within the virtual server to respond to client queries, issue heartbeat messages, and respond to terrain mutating events. Doing so would be both wasteful of network bandwidth and would impose unnecessary burden on the clients. Within the virtual server, a single server will be designated as the *master*. It is the master server only which responds to events which mutate the terrain surface, issues TerrainPatchUpdatePDUs, heartbeat messages, and responds to client queries for the

Figure 4.1: Virtual Terrain Server Concept

Client Simulations



purpose of checksum resolution and updating late joining entities. Passively monitoring network activity within the virtual server are one or more *slave* servers. If the master server were to fail, the slave servers would employ a leader election algorithm³⁷ to elect a server to take over as the new master. Because it is expected that each server have a unique DIS entity identifier, the leader election algorithm may be as simple as selecting the server with the next higher entity address. In the case that the server with the highest identifier fails, the server with the lowest entity identifier would become the master.

The servers which comprise the virtual server may not reside on the same local area network. This is to improve fault tolerance for distributed simulations conducted over a wide area network such as the Defense Simulation Internet. When designing the implementation of the virtual server for a wide area network, we can no longer assume that all servers within the virtual server will receive packets in the same order.³⁸ This imposes some complications which we examine later.

4.2 Consistency Among Servers

A major complication imposed by the existence of multiple redundant servers is the fact that each server maintains its own local copy of the terrain database in its current form. It is imperative that each of the servers have a consistent view of the terrain. If a slave server were to take over as the master at some point, a lack of consistency could lead to unexpected changes in the terrain. To illustrate this, consider the following scenario: The master server has a patch of terrain containing a crater caused by a large bomb detonating at the surface of the terrain. The slave server's version of this patch however does not have the crater for that particular patch of terrain. The master server fails at some point in the future, and the slave takes over as the new master. If another simulation event caused a mutation to the same patch of terrain, the original crater will disappear when the new master server transmits a terrain update message for that patch of terrain.

An intuitive solution to the consistency problem is to allow the servers to process terrain mutating events in parallel, updating their own local copies of the terrain database in

response to terrain mutating simulation events. At first glance, this would appear to work if all the servers were running identical versions of the software. The problem with this approach is two-fold:

1. If servers reside on different local area networks, they may not receive events in the same order. Applying events to a patch of terrain in different orders is likely to produce different results.
2. The master server may miss a particular event which is received and processed by the slaves. This would result in inconsistencies between the master and slave servers.

The above discussion has illustrated that the master server only should be responsible for maintaining the terrain database. The correct order in which events are applied to the terrain is defined as the order in which the events are received and processed by the master. If an event is missed by the master, any effects the event would have had on the terrain are simply lost. Slave servers will update their local copies of the terrain database in the same manner as client simulations - through the receipt of TerrainPatchUpdatePDUs issued by the master. Like clients, slave servers will use the checksum provided in the heartbeat message to verify that their local copy of the terrain database is consistent with the master server.

The following discussion identifies two degrees of fault tolerance which we will refer to as Level I and Level II. The two levels of fault tolerance are defined and then the argument is made why Level II fault tolerance is not practical given the current state of networking technology.

4.3 Level I Fault Tolerance

The Level I virtual server consists of a single master server and one or more slave servers. Initially, the master server is in control. The slaves behave as clients by using the Checksum Tree Protocol to maintain terrain database consistency with the master server. Slave servers do not attempt to process any terrain mutating events and discard such events if they are received. The master is expected to issue heartbeat messages at some predetermined interval as per the Checksum Tree Protocol. The absence of heartbeat messages over some time window indicates to the slaves that the master has failed, and a new master should be elected from among the slaves. When the elected slave takes over as the new master, it will begin issuing heartbeat messages and processing terrain mutating events.

Level I fault tolerance is vulnerable between the time in which the master server fails and a new master is elected. During this period, there is no server on the network behaving as the master. The effects upon the terrain of any terrain mutating events occurring during this period would be lost, and the simulation would temporarily degenerate to a static terrain simulation.

Another complication arises if the master and slaves reside on separate local-area networks connected by gateways. The temporary saturation or failure of a router may lead a slave to incorrectly infer that the master has failed. This could result in multiple master servers. The virtual server would have to include a resolution protocol to deal with this condition.

4.4 Level II Fault Tolerance

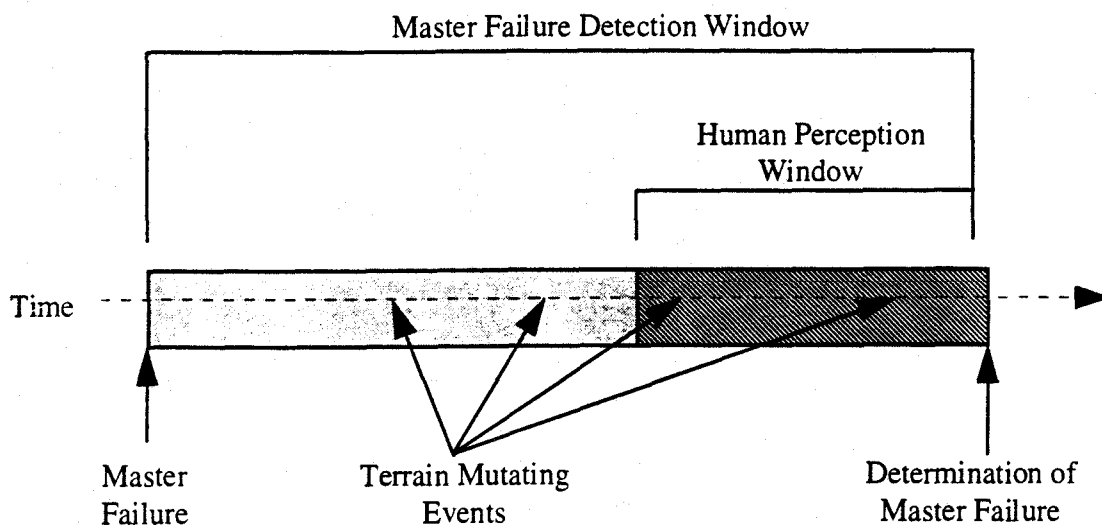
Level I fault tolerance was shown to be vulnerable during the interval between when a master fails and a new master is elected. A higher degree of fault tolerance may be achieved by requiring the slave server to buffer terrain mutating events until it has been confirmed by the slave servers that the master has processed these events and has issued TerrainPatchUpdatePDUs accordingly. If the master server were to fail after receiving terrain mutating events but before issuing the corresponding TerrainPatchUpdatePDU, the newly elected master could take over without the loss of the effects of these events as these events have been saved by the slave server.

An important factor to consider in man-in-the-loop simulations is the human perception of latency. It would be undesirable for events which mutate the terrain to have effects which are excessively delayed. It would be preferred that such events have no effect on the terrain at all, temporarily degenerating the simulation to a static terrain simulation. Level II fault tolerance requires slaves to save terrain mutating events so that they may be applied to the terrain in the event of a master server failure. However if the servers which comprise the virtual server are physically located on separate local area networks, the time in which a slave may infer that a master server has failed is on the order of seconds.³⁹ Figure 4.2 illustrates the temporal relationships between the failure of a master server, the determination that the master server has failed, and the arrival of terrain mutating events in the interim. As a slave takes over as the new master, the slave should not apply buffered

terrain mutating events which fall outside the human perception window, for human participants will perceive the delayed effects of these events.

Because the human perception window is on the order of tens of milliseconds in length,⁴⁰ and the master server failure detection window is on the order of seconds, it is unlikely that the added complexity of a Level II fault tolerant virtual server is justified.

Figure 4.2



5 Details of Implementation

This chapter describes the details of implementation of the Checksum Tree Protocol and the Virtual Server. The implementations are described in terms of software objects used in the design and the protocol data units to be added to the DIS standard.

5.1 The TerrainDB Class

The TerrainDB class is used to provide an object-oriented⁴¹ interface to the dynamic distributed terrain database. Clients use the TerrainDB class to interface with the dynamic terrain server and also to acquire the terrain feedback information necessary to conduct their simulation. Figure 5.1 illustrates the TerrainDB class specification.

The class constructor requires the path-name of the file which contains the initial state of the terrain database. Dynamic mutations to the terrain database are accomplished by the *processNetUpdate* method. When a client simulation receives a TerrainPatchUpdatePDU over the network interface, the PDU is handed off to the TerrainDB object for processing via a call to this method.

The revision number of a particular patch of terrain may be queried via the *getRevisionNumber* method. Internally, this method is used by the TerrainDB class to avoid processing repeated TerrainPatchUpdatePDUs for the same patch and revision number combination. Such a TerrainPatchUpdatePDU may be received if the server retransmits a patch update for a client who missed the original broadcast. The servers

use this method for generating the next higher revision number when sending out TerrainPatchUpdatePDUs for newly modified terrain patches.

The remaining methods in this class pertain to the actual simulation of a vehicle. The *placeVehicle* class orients a ground vehicle on the terrain surface given a position, heading, and speed. This method returns the elevation component of the position, the velocity vector of the vehicle, and the three Euler angles of rotation required to rotate the vehicle from body coordinates to world coordinates. The *getElevation* method returns the elevation of the terrain at a particular location. The *getPatchElevations* method returns the entire grid of elevation samples for a particular patch of terrain. This method is used primarily by servers for the purpose of calculating terrain profiles in response to simulation events capable of mutating the terrain. .

Figure 5.1

```
class TerrainDB
{
public:
    TerrainDB( char *fileName );
    ~TerrainDB();
    void placeVehicle(
        double location[3], // X-Y in, Z out
        double heading, // Radians clockwise relative to north(in)
        double speed, // Meters/second (in)
        double velocity[3], // Meters/second (out)
        double eulers[3]); // Angles of rotation (out)

    void getElevation( double location [3] ); // X-Y in, Z out
    double getElevation( double x, double y );
    void processNetUpdate( TerrainPatchupdatePDU *pdu );
    int getRevisionNumber( int patchCoordX, int patchCoordY );
    int getPatchNumElevationSamples( int patchCoordX, int patchCoordY );
    int getPatchSampleSize( int patchCoordX, int patchCoordY );
    void getPatchElevations( int patchCoordX, int patchCoordY, float *buf );

private:
    ...
};
```

5.2 The Checksum Tree Class

Both client simulations and terrain servers need to maintain a data structure representing the state of their local versions of the Checksum Tree. This is facilitated by the Checksum Tree Class illustrated in Figure 5.2.

Both the clients and the servers create an instance of the ChecksumTree class for each active terrain database in the simulation. The ChecksumTree class constructor requires an integer specifying how many levels deep to create the tree. This value could be provided in the terrain database header or computed from the size of the terrain database.

The *setValue* method allows the class user to set the revision level for a particular leaf of the tree. Within the *setValue* method, the ChecksumTree is recursively ascended⁴² computing the new checksums at each level until the root node is reached.

The *getChecksums* method allows the class user to query a particular node of the checksum tree for the checksums of the node's children. This is used in the checksum resolution process by the clients to determine which nodes of the checksum tree disagree

Figure 5.2

```
class ChecksumTree
{
public:
    ChecksumTree( int numLevels );

    --ChecksumTree();

    void setValue( int x, int y, unsigned long value );

    void getchecksums( int numLevels, unsigned short *nodeIds,
        unsigned long checksums[100] );

    unsigned long getChecksum() { return checksum; }

private:
    unsigned long checksum;
    int level;

    ChecksumTree    *children[100];
};
```

Figure 5.3

```
struct TerrainPatchUpdatePDU
{
    PDUHeader          header;
    SimulationAddress  server;
    unsigned short     terrainDatabaseID;
    unsigned short     revisionNumber;
    unsigned short     patchCoordX;
    unsigned short     patchCoordY;
    short              sampleSize;
    short              unused;
    float              zValues[1]; // Varies
}
```

with the server's checksum tree.

The *getChecksum* method returns the overall checksum for the entire checksum tree. This is used by clients to compare their database checksum with the checksum indicated by the terrain server in the heartbeat message.

5.3 DIS Protocol Extensions

The Checksum Tree Protocol requires a minimal amount of additions to the DIS protocol. The TerrainPatchUpdatePDU illustrated in Figure 5.3 is the mechanism by which an update to a patch of terrain is communicated to simulation clients. For simplicity and illustration purposes, this PDU represents the terrain within the patch by a grid of elevation samples. Better terrain representation schemes may be chosen, but this is beyond the scope of this thesis. Also, no cultural or feature data is present in this implementation.

The terrain server periodically broadcasts "heartbeat" messages which contain the

Figure 5.4

```
struct TerrainHeartbeatPDU
{
    PDUHeader          header;
    SimulationAddress  server;
    unsigned long       databaseChecksum;
    unsigned short     terrainDatabaseID;
};
```

Figure 5.5

```
#define MaxChecksumTreeLevels 10
struct ChecksumTreeQueryPDU
{
    PDUHeader          header;
    SimulationAddress  client;
    unsigned short     levelsPresent;
    unsigned short     nodeIds[MaxChecksumTreeLevels];
}
```

root checksum for the terrain database. The TerrainHeartbeatPDU illustrated in Figure 5.4.

Upon receipt of a TerrainHeartbeatPDU, client simulations compare the database checksum indicated by the PDU with the root checksum of their local instance of the ChecksumTree class. If the two values disagree, the client simulation has missed one or more TerrainPatchUpdatePDUs. The client must then query the server for the checksums of the root's children to determine which patch of terrain is stale. This is accomplished via the ChecksumTreeQueryPDU illustrated in Figure 5.5.

Upon receipt of a ChecksumTreeQueryPDU, the server must respond with children's checksums of the requested node. This is accomplished via the ChecksumTreeResponsePDU illustrated in Figure 5.6. Both the ChecksumTreeQueryPDU and the ChecksumTreeResponsePDU identify the particular node of the checksum tree by the *nodeIds* field. This field contains an array of *nodeIds*. The index into this array corresponds to the level of depth into the checksum tree. The *levelsPresent* field specifies how many levels of depth is contained in the array.

Figure 5.6

```
struct ChecksumTreeResponsePDU
{
    PDUHeader          header;
    SimulationAddress  client;
    unsigned short     levelsPresent;
    unsigned short     nodeIds[MaxChecksumTreeLevels];
    unsigned long      checksums[100];
};
```

The initial query issued by a client will contain 0 for *levelsPresent*. This indicates that the client wishes to acquire the checksums of the root's immediate children. Upon receipt of the *ChecksumTreeResponsePDU* from the server, the client may determine which of the server's root child checksum disagrees with the client's local version of the checksum tree. When a disagreement is found, another *ChecksumTreeQueryPDU* will be issued by the client. This time, *levelsPresent* will contain 1 and *nodeIds[0]* will contain the index of the node which disagreed. This process will continue until the server receives a *ChecksumTreeQueryPDU* for a leaf of the checksum tree. The server will respond by transmitting the *TerrainPatchUpdatePDU* for the patch of terrain which is represented by the particular leaf node of the checksum tree.

5.4 Client-Side Procedural Code

This section provides an example of procedural code used by clients for the implementation of the Checksum Tree Protocol. It is assumed that a client simulation will periodically (possibly at the beginning of each simulation frame) process all inbound PDUs. As the simulation loops through all pending inbound PDUs, various routines will be dispatched depending on the type of PDU received. This section illustrates a possible implementation of the dispatch routines for the Checksum Tree Protocol PDUs.

Upon receipt of a *TerrainPatchUpdatePDU* from the server, the client hands off the PDU to the *TerrainDB* object for processing. The corresponding leaf node of the checksum

tree is set to the revision number of the patch. This is accomplished by the *processTerrainUpdatePDU* procedure illustrated in Figure 5.7.

Figure 5.7

```
static TerrainDB *tdb;           // Dynamic Terrain database object

static ChecksumTree *cstree;    // Checksum Tree object

void processTerrainUpdatePDU( TerrainPatchUpdatePDU *pdu )
{
    tdb->processNetUpdate(pdu);
    cstree->setValue(pdu->patchCoordX, pdu->patchCoordY, pdu->revisionNumber);
}
```

Upon receipt of a heartbeat message from the server, the client will invoke the *processHeartbeatPDU* procedure illustrated in Figure 5.8. This procedure queries the checksum tree object for the root checksum. *cstree* is a pointer to the root node of the checksum tree object. If the root checksum disagrees with the checksum indicated by the *TerrainHeartbeatPDU*, a *ChecksumTreeQueryPDU* is formatted and broadcasted onto the network. The *levelsPresent* field of the *ChecksumTreeQueryPDU* is set to 0 to indicate that the client is interested in the checksums of the root's children. This initial query bootstraps the resolution process. Upon the receipt of a *ChecksumTreeResponsePDU* from the server, the client is able to determine if any disagreement exists between the client's version of the checksum tree and the server's version. This is accomplished by the *processCTResponsePDU* procedure illustrated in Figure 5.9.

Figure 5.9

```
void processCTResponsePDU( ChecksumTreeResponsePDU *pdu )
{
    // Make sure we are interested in this response
    if (pdu->client.site != site || pdu->client.host != host)
        return;

    // Get our version of the checksums
    unsigned long checksums[100];
    cstree->getChecksums(pdu->levelsPresent, pdu->nodeIds, checksums);

    // Find the one which disagrees with the server
    for (int i = 0; i < 100; i++)
        if (checksums[i] != pdu->checksums[i])
            break;

    if (i < 100)
    (
        // Format and spew another query message
        ChecksumTreeQueryPDU qpdu;

        qpdu.header.version = DISProtocolVersionCurrent;
        qpdu.header.exercise = 1;
        qpdu.header.kind = ChecksumTreeQueryPDUKind;
        qpdu.client.site = site;
        qpdu.client.host = host;
        qpdu.levelsPresent = pdu->levelsPresent + 1;

        for (int j = 0; j < pdu->levelsPresent; j++)
            qpdu.nodeIds[j] = pdu->nodeIds[j];
        qpdu.nodeIds[pdu->levelsPresent] = i;

        dis.sendPacket((char *)&qpdu, sizeof(qpdu));
    )
};
```

The first step in this procedure is to determine if this response packet is directed towards the client in question. If this is the case, the checksum tree object is queried for the checksums corresponding to the node indicated by the ChecksumTreeResponsePDU. Each

Figure 5.8

```
void processHeartbeatPDU( TerrainHeartbeatPDU *pdu )
{
    if (cstree->getChecksum() != pdu->databaseChecksum)
    {
        // Issue a query message to the server
        ChecksumTreeQueryPDU qpdu;

        qpdu.header.version = DISProtocolVersionCurrent;
        qpdu.header.exercise = 1;
        qpdu.header.kind = ChecksumTreeQueryPDUKind;
        qpdu.client.site = site;
        qpdu.client.host = host;
        qpdu.levelsPresent = 0;

        dis.sendPacket((char *)&qpdu, sizeof(qpdu));
    }
};
```

checksum is compared to that indicated by the response packet. If any disagreements are found, a `ChecksumTreeQueryPDU` is formatted to query for the next level of depth into the checksum tree.

Note that the Checksum Tree Protocol is stateless in the sense that `ChecksumTreeQueryPDUs` and `ChecksumTreeResponsePDUs` contain all the information necessary for the clients and servers to react accordingly. Neither the clients nor the servers are required to maintain local state information regarding the state of the resolution process. This greatly simplifies the implementation on both the client and server sides. Furthermore, acknowledgments are not necessary for these PDUs. If a `ChecksumTreeQueryPDU` or a `ChecksumTreeResponsePDU` were to be lost, the resolution process would be re-initiated on the receipt of the next heartbeat message from the server because the root checksums would still be in disagreement. This fact suggests a criteria for the selection of the interval between heartbeat messages issued by the server. The interval should be large enough to allow for the entire resolution process between a client and a server to be completed.

5.5 Server-Side Procedural Code

Within the terrain server itself, a single procedure is necessary to implement the Checksum Tree Protocol. This procedure is invoked upon the receipt of a `ChecksumTreeQueryPDU`. The *processCTQueryPDU* is illustrated in Figure 5.10.

The first step in this procedure is to determine if the query is for a leaf node in the Checksum Tree. If the query is for a leaf node, the server will respond by transmitting a

TerrainPatchUpdatePDU containing the information for the patch of terrain represented by the indicated leaf of the Checksum Tree. If the query is not for a leaf node, the server will respond by transmitting a ChecksumTreeResponsePDU containing the checksums of the children of the indicated node.

If the *levelsPresent* field of the ChecksumTreeQueryPDU is equal to the depth of the ChecksumTree, the query is for a leaf node. The server must then determine the amount of memory required to store the TerrainPatchUpdatePDU by querying the TerrainDB object for the sample interval of the indicated patch. After memory is allocated to store the PDU, the TerrainDB object is queried for the elevation samples of the indicated patch. After the PDU header information is supplied, the packet is transmitted onto the network.

If the *levelsPresent* field of the ChecksumTreeQueryPDU is less than the depth of the Checksum Tree, then the server must format and transmit a ChecksumTreeResponsePDU. These PDUs are fixed in size, so memory is allocated off the stack by the instantiation of a local variable. After filling out the appropriate header information, the Checksum Tree object is queried for the checksums of the node indicated by the ChecksumTreeQueryPDU. The formatted PDU is then broadcasted onto the network.

Figure 5.10

```
void processCTQueryPDU( ChecksumTreeQueryPDU *query )
{
    if (query->levelsPresent == ChecksumTreeDepth)
    {
        // Client has identified stale patch - send them a new terrain update

        int    patchCoordX = query->nodeIds[0] - (query->nodeIds[0]%10) +
            (query->nodeIds[1]/10);
        int    patchCoordY = 10*(query->nodeIds[0]%10) + (query->nodeIds[1]%10);

        int    numSamples = tdb->getPatchNumElevationSamples(patchCoordX,
            patchCoordY);

        int    pduSize = sizeof(TerrainPatchUpdatePDU) +
            sizeof(float)*numSamples;
        TerrainPatchUpdatePDU *pdu = (TerrainPatchUpdatePDU *)
            new char[pduSize];

        pdu->header.version = DISProtocolVersionCurrent;
        pdu->header.exercise = 1;
        pdu->header.kind = TerrainPatchUpdatePDUKind;
        pdu->server = me;
        pdu->terrainDatabaseID = 1;
        pdu->patchCoordX = patchCoordX;
        pdu->patchCoordY = patchCoordY;
        pdu->revisionNumber = tdb->getRevisionNumber(patchCoordX, patchCoordY);
        pdu->sampleSize = tdb->getPatchSampleSize(patchCoordX, patchCoordY);
        tdb->getPatchElevations(patchCoordX, patchCoordY, pdu->zValues);

        printf("Sending TerrainPatchUpdatePDUKind to update client\n");
        dis.sendPacket((char *)pdu, pduSize); // Send update to clients
        delete pdu;
    }
    else
    (
        // Client needs to go one level deeper into the checksum tree
        ChecksumTreeResponsePDU response;

        response.header.version = DISProtocolVersionCurrent;
        response.header.exercise = 1;
        response.header.kind = ChecksumTreeResponsePDUKind;
        response.client.site = query->client.site;
        response.client.host = query->client.host;
        response.levelsPresent = query->levelsPresent;

        for (int i = 0; i < query->levelsPresent; i++)
            response.nodeIds[i] = query->nodeIds[i];

        cstree->getChecksums(response.levelsPresent, response.nodeIds,
            response.checksums);

        printf("Sending ChecksumTreeResponsePDU to client\n");
        dis.sendPacket((char *)&response, sizeof(response));
    }
};
```

5.6 Virtual Server Implementation

A single executable serves as both the master and slave servers. The flag within the server indicates to the server code whether the server is operating in master or slave modes. In slave mode, the server is simply a client simulation listening to the network for TerrainPatchUpdatePDUs and Checksum Tree Protocol PDUs to keep its local version of the terrain database consistent with the current master server. The slave server also expects to receive TerrainHeartbeatPDUs from the current master at some predetermined interval. If the slave fails to receive a TerrainHeartbeatPDU within some time-out period, the slave will assume the master server is off-line and initiate a protocol to become the new master.

The master server listens to the network for events capable of mutating the terrain and also listens for ChecksumTreeQueryPDUs from client simulators. If a DetonationPDU is received, the master determines if any of patches of terrain were affected. If so, a TerrainPatchUpdatePDU is sent out for each affected patch. The master server also transmits a periodic TerrainHeartbeatPDU which contains the checksum for the root node of the checksum tree and informs slave servers that the master is on-line.

The implementation of the virtual server outlined above requires that certain functions be scheduled for invocation in the future. The *SchedQueue* object illustrated in Figure 5.11 is the mechanism by which this is accomplished. The *SchedQueue* object has several advantages over the UNIX interval timer mechanisms. The *SchedQueue* object provides a single-threaded approach to providing time-out functions. The UNIX interval timer mechanisms invoke time-out functions in the context of a signal handler.⁴³ Because of

Figure 5.11

```
#ifndef SchedQueue_h
#define SchedQueue_h

#include <stdio.h>

typedef unsigned long SchedEventId;

typedef void (*SchedFunc)( void *arg );

class SchedQueue
{
public:
    SchedQueue();
    ~SchedQueue();

    void tick( double currentTime );

    SchedEventId addEvent( SchedFunc func, void *arg, double time );

    void removeEvent( SchedEventId id );

private:
    struct SchedEvent
    {
        SchedFunc      func;
        void           *arg;
        double         time;
        SchedEventId  id;
        SchedEvent    *nextEvent;
    };

    SchedEvent *queue;
};

#endif
```

this, applications may not access local data structures within the signal handlers for concurrency problems may arise unless complex concurrency handling code is employed. Another disadvantage to the UNIX interval timer mechanism is that the resolution of the timer is limited to the tick rate of the operating system which is typically 60Hz to 100Hz for most popular systems. A final advantage of the *SchedQueue* object is that it does not require context switching for the time-out functions to be invoked, thereby making it more efficient.

The *addEvent* method of the *SchedQueue* object allows the class user to schedule the invocation of a function which has pointer to type void as a argument. The *tick* method is

invoked by the class user at the top of each simulation frame. It is during the call to tick that any functions which have timed-out are executed. The removeEvent method allows the class user to remove a scheduled function invocation which was previously scheduled by a call to addEvent.

The main routine of the virtual server is illustrated in Figure 5.12. The first step is to determine whether the server is to be initially configured as the master or a slave. The second command line argument determines the initial mode of the server (e.g. M for master, or S for slave).

The next step involves determining the server address from the third command line argument. The server address corresponds to the SimulationAddress from the DIS protocol which is a integer tuple specifying a unique id for the site and host computer.

Figure 5.12

```
main( int argc, char **argv )
(
    if (argc != 4)
        usage(argv[0]);

    // To be the master, or not to be the master
    switch (argv[2][0])
    (
    case 'M': master = 1; break;
    case 'S': master = 0; break;
    default: usage(argv[0]);
    }

    // Server address
    int site, host;
    if (sscanf(argv[3], "%d-%d", &site, &host) != 2)
        usage(argv[0]);
    me.site = site;
    me.host = host;

    // Open the terrain database
    tdb = new TerrainDB(argv[1]);

    // Create a checksum tree
    cstree = new ChecksumTree(3);

    // Start sending heartbeat messages if we are the master, otherwise we
    // are a slave and will give the master 10 seconds to tell us he is alive
    if (master)
        sendHeartbeat();
    else
        becomeMasterEventId = queue.addEvent(becomeMasterEventFunc, 0, 10.0);

    // Loop forever...
    while (1)
    (
        simTime = getTime();
        queue.tick(simTime);           // Tick the event scheduler
        flushNetwork();               // Process PDUs
    )
}
```

Both the master and slave servers create an instance of the TerrainDB and a ChecksumTree objects. The initial state of the terrain database is loaded from a dynamic terrain database file which is indicated by the first command line argument.

If the server is being initialized as the master, the *sendHeartbeat* function (see Figure 5.13) is called which transmits the initial TerrainHeartbeatPDU and schedules itself for future invocations. If the server is being initialized as a slave, the *becomeMasterEventFunc* (see Figure 5.13) is scheduled for invocation in the future. If a TerrainHeartbeatPDU is

Figure 5.13

```
void sendHeartbeat()
{
    TerrainHeartbeatPDU pdu;

    pdu.header.version = DISProtocolVersionCurrent;
    pdu.header.exercise = 1;
    pdu.header.kind = TerrainHeartbeatPDUKind;
    pdu.server = me;
    pdu.databaseChecksum = cstree->getChecksum();
    pdu.terrainDatabaseID = 1;

    printf("Sending heartbeat message\n");
    dis.sendPacket((char *)&pdu, sizeof(pdu));

    // Schedule next invocation
    heartbeatEventId = squeue.addEvent(
        (SchedFunc)sendHeartbeat, 0, simTime+HeartBeatInterval);
}

static void becomeMasterEventFunc( void* )
{
    printf("*** Becoming Master Server ***\n");
    master = 1;
    sendHeartbeat();
}
```

received by the slave, this function is removed from the *SchedQueue* object and rescheduled for further into the future. This is the mechanism by which slave servers detect the absence of a master.

Once during every frame of the simulation a call is made to the *flushNetwork* routine illustrated in Figure 5.14. The server is interested in different sets of inbound PDUs depending on the current operating mode of the server. If the server is operating in slave mode, the server is basically acting as a client simulator by using the Checksum Tree Protocol to ensure its local copy of the terrain database is consistent with the master's. Any time a PDU is received from the master server, the *masterIsAlive* routine is called which defers the scheduled event for the slave to become the master.

The master server responds to DetonationPDUs by calling *processDetonation*. This route assesses any terrain damage and transmits TerrainPatchUpdatePDUs accordingly. If

the master server receives a TerrainHeartbeatPDU from another master, the server yields to the other master and becomes a slave server. If the master receives a ChecksumTreeQueryPDU from a client simulator, it must respond by issuing a ChecksumTreeResponsePDU or a TerrainPatchUpdatePDU. This is accomplished by the *processCTQueryPDU* illustrated in Figure 5.10.

Figure 5.14

```
static void flushNetwork()
(
    char buffer(1500);

    // Read all PDUs which have accumulated since the last frame
    while (dis.readPacket(buffer, sizeof(buffer)) > 0)
    {
        PDUHeader *header = (PDUHeader *)buffer;

        if (master)
        {
            // The master handles DetonationPDUs and ChecksumTreeQueryPDUs.
            // If we receive a heartbeat from another master, we yield to them

            switch (header->kind)
            (
            case DetonationPDUKind:
                processDetonation((DetonationPDU *)buffer);
                break;
            case ChecksumTreeQueryPDUKind:
                processCTQueryPDU((ChecksumTreeQueryPDU *)buffer);
                break;
            case TerrainHeartbeatPDUKind:
                TerrainHeartbeatPDU *pdu = (TerrainHeartbeatPDU *)buffer;
                if (pdu->server.site != me.site || pdu->server.host != me.host)
                (
                    // Someone else thinks they are the master, so yield to them
                    master = 0;
                    becomeMasterEventId = queue.addEvent(
                        becomeMasterEventFunc, 0,
                        simTime + 2.0*HeartBeatInterval);
                    queue.removeEvent(heartbeatEventId);
                )
            )
        }
        else
        (
            // Slaves basically act as clients, getting terrain updates from
            // the master

            switch (header->kind)
            {
            case TerrainPatchUpdatePDUKind:
                processTerrainUpdatePDU( (TerrainPatchUpdatePDU *)buffer );
                masterIsAlive();
                break;
            case TerrainHeartbeatPDUKind:
                processHeartbeatPDU( (TerrainHeartbeatPDU *)buffer );
                masterIsAlive();
                break;
            case ChecksumTreeResponsePDUKind:
                processCTResponsePDU( (ChecksumTreeResponsePDU *)buffer );
                masterIsAlive();
                break;
            )
        )
    }
}
```

6 Testing the Implementation

This chapter describes the testing methodology and results for the Checksum Tree Protocol and the Virtual Server. Testing the Checksum Tree Protocol involves the interaction of four distributed applications: a stealth device, a ground vehicle simulation, a dynamic terrain server, and a PDU generation tool. Each of these applications will be described in turn. Three different testing scenarios will be described followed by a discussion of the results of each scenario.

6.1 The Stealth Simulator

In order to verify the effectiveness of the Checksum Tree Protocol, some mechanism must be employed to provide a real-time three-dimensional view into the virtual world created by a distributed interactive simulation. This device must be capable of rendering the terrain and features of the virtual world, and also any vehicles participating in the simulation. A *Stealth* simulator is such a device. A Stealth is referred to as such because it is a passive observing participant in the simulation. The Stealth view itself does not correspond to any particular vehicle in the simulation, and does not generate any EntityStatePDUs. For this reason, a Stealth is invisible to other simulators. Features of a typical Stealth include the following:

- The capability to "teleport" to any location within the terrain database and assume any orientation.

- The capability to tether the viewport to any given vehicle in the simulation
- The capability to "free fly" about the terrain database.

A Stealth simulator was developed as an independent study project in conjunction with this thesis. The *Rybacki Stealth* requires a Silicon Graphics workstation equipped with a hardware z-buffer. As tested on the SGI Crimson Extreme platform (50mhz R4000), the Stealth was capable of maintaining real-time frame rates (> 15Hz) with a 20 degree field of view, a terrain horizon of 5 kilometers, and several vehicles within the field of view. Figure 6.1 illustrates a typical Stealth view with terrain and vehicles on the Ft. Hunter-Ligget terrain database.

6.2 The Truck Simulator

A ground vehicle simulator was also developed to participate in the testing. Ground vehicles must correctly follow the terrain surface in a simulation. The Truck simulator is a workstation-based simulation which enables one to drive a truck about the terrain database. This simulator does not attempt to emulate the dynamics of any real vehicle, its sole purpose is to illustrate the effectiveness of the Checksum Tree Protocol. The Truck simulator supports the following commands:

<u>Command</u>	<u>Function</u>
s N	Set the speed of the vehicle to N meters per second
r	Turn right a little
R	Turn right a lot
l	Turn left a little
L	Turn left a lot
q	Exit the simulation

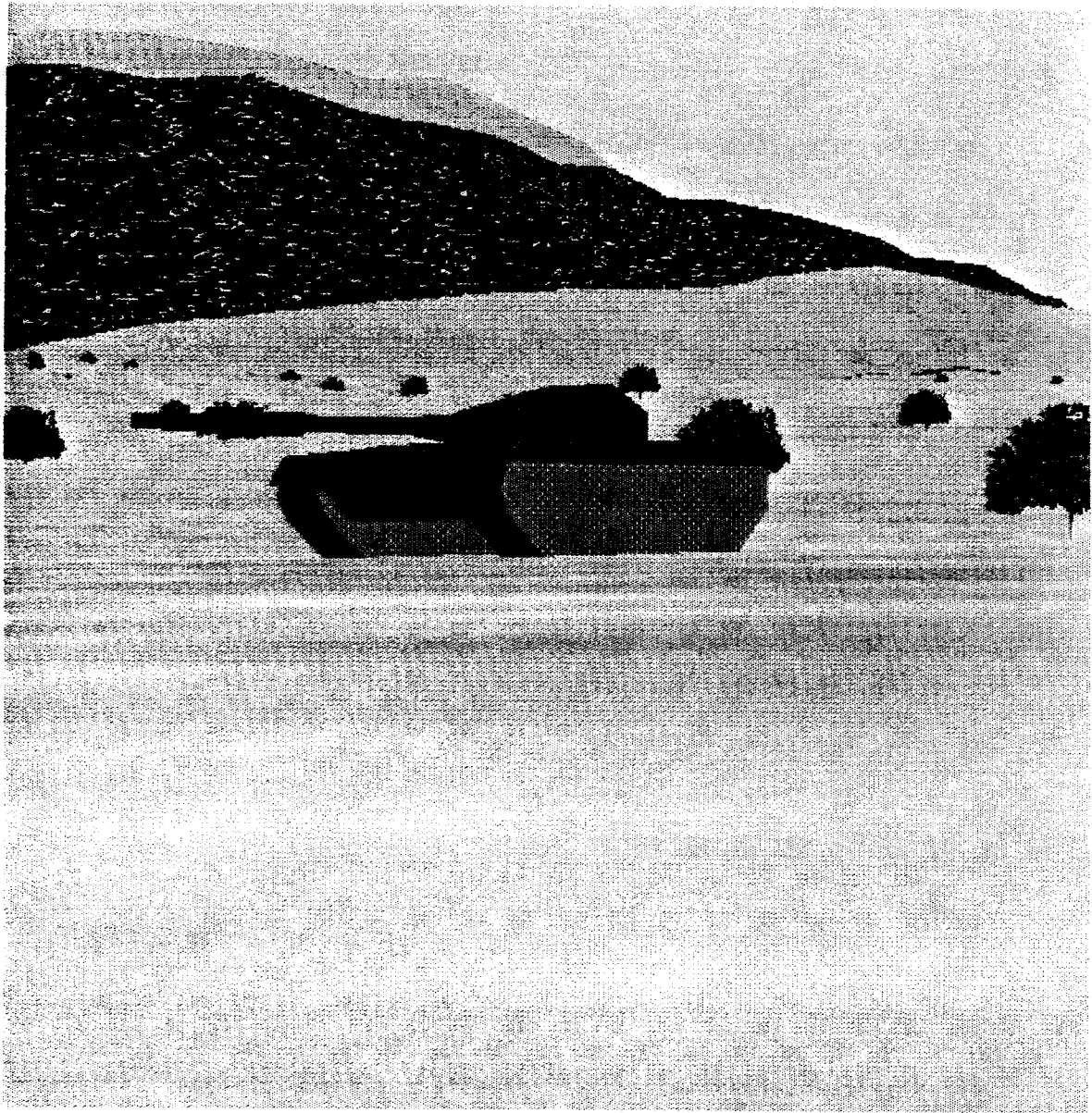


Figure 6.1 A Stealth view of terrain and vehicles on the Ft. Hunter-Ligget terrain database

6.3 The Dynamic Terrain Server

The dynamic terrain server monitors the network awaiting events which are capable of modifying the terrain. The detonation of a munition is an example of such an event. If a detonation is sufficiently close to the terrain surface, the terrain server will create a crater in the terrain under the point of detonation and broadcast a TerrainPatchUpdatePDU to update the simulation clients. Note that a realistic implementation of a dynamic terrain server would take into account the type of warhead and soil attributes in the computation of terrain damage. This level of modeling is beyond the scope of this thesis. The simplistic terrain server illustrated here is adequate for the purposes of illustrating the concepts presented in this thesis.

6.4 The PDU Generation Tool

PDUGEN is a tool which generates arbitrary PDUs and broadcasts them onto the DIS network. PDUGEN reads an ASCII file which describes PDU structures using a simple data-notation language. The user is presented with the fields of the PDU for which PDUGEN is to generate. PDUGEN provides controlled stimulus to the dynamic terrain server by generating DetonationPDUs in the absence of an actual simulator capable of doing so.

The testing scenarios involve orienting the Stealth viewport so that the Truck simulator is within the field of view. A DetonationPDU is issued by PDUGEN which results in a crater in the pathway of the truck. The successful receipt of a TerrainPatchUpdatePDU

by the Stealth simulator manifests itself as a crater appearing in the appropriate location in the terrain. In the case of the Truck simulator, successful receipt of a TerrainPatchUpdatePDU manifests itself as the truck correctly following the terrain surface into the crater, and is visualized from the Stealth.

6.5 Checksum Tree Testing Scenarios

Three testing scenarios were devised to test the Checksum Tree Protocol. The first form involves contriving a condition where at least one of the simulation clients misses a TerrainPatchUpdatePDU, forcing the use of Checksum Tree Protocol to become updated. The second form of testing involves bringing a simulator on-line after the server has modified the dynamic terrain database. The final form of testing involves restarting the dynamic terrain server mid-course in a simulation.

6.5.1 Scenario I

In order to exercise the Checksum Tree Protocol, at least one of the simulators must miss the TerrainPatchUpdatePDU which is issued by the server at the point of detonation. Because it is difficult to contrive such a condition, the terrain server code was modified so that the TerrainPatchUpdatePDU would not be sent out at the point of detonation. This guarantees that both the Stealth simulator *and* the Truck simulator miss the TerrainPatchUpdatePDU, which was actually never transmitted. The two client simulators would become aware of this condition on the receipt of the next TerrainHeartbeatPDU from

the terrain server because the database checksums disagree. Both clients then initiate the resolution process using the Checksum Tree protocol.

6.5.2 Scenario II

Another form of testing involves bringing the simulators on line after the terrain server has received a DetonationPDU and transmitted a TerrainPatchUpdatePDU accordingly. Upon receipt of the first TerrainHeartbeatPDU from the server, the clients initiate the resolution process with the server and become updated.

6.5.3 Scenario III

The final form of testing involves restarting the terrain server mid-course in a simulation, after the terrain has been modified. Although not practical in a real simulation, this test exercises the Checksum Tree protocol and illustrates its effectiveness. Our model assumes that the unique correct state of the terrain is the one held by the terrain server. When the server comes on-line after a modification, the clients "regress" to the original state of the terrain.

6.5.4 Checksum Tree Testing Results

The results of Scenario I showed that the Checksum Tree protocol was successful in bringing the clients up-to-date in an efficient manner. There was little perceived latency between the time of detonation and the time in which the crater appeared on the Stealth. At

what was perceived to be same instant in time, the Truck simulator began to follow the updated terrain surface into the crater.

. Since human perception of latency is an important issue in man-in-the-loop simulations, the interval between heartbeat messages from the terrain server must be chosen to be sufficiently small. This parameter imposes a lower bound on the latency between the simulation event which mutates the terrain, and the time in which the updated terrain information is available to the simulator in the event the initial broadcast of the terrain update message is lost. The latency is less of an issue for simulators which are not operating geographically close to the point of terrain mutation and hence are not currently using the affected patch.

An interesting discovery was made while testing the Checksum Tree protocol as described above. Since both clients miss the initial TerrainPatchUpdatePDU, they both initiate the resolution process on the receipt of the next TerrainHeartbeatPDU from the server. However, only one client carries the resolution process to full course. The first client which elicits a retransmission of the TerrainPatchUpdatePDU from the server benefits all other clients which missed the same initial TerrainPatchUpdatePDU. As the slower clients receive and process the TerrainPatchUpdatePDU from the server, their Checksum Trees become consistent with the server's causing the resolution process to be terminated early. Figure 5.9 illustrates the *processCTResponsePDU* procedure used by the clients. When the server responds with a ChecksumTreeResponsePDU for a client's query, the checksums contained in the server's message become consistent with the client's local copy

of the Checksum Tree, terminating the resolution process. This feature of faster clients benefiting slower clients when the same TerrainPatchUpdatePDU is missed will be particularly useful in wide-area distributed simulations where an entire leg of the network is momentarily off-line, causing multiple clients to miss the same TerrainPatchUpdatePDUs.

The results of testing Scenario II were equally impressive. The Stealth simulator was started after the time of detonation. When the Stealth rendered its initial view of the terrain, the crater was already visible. Likewise, initializing the Truck simulator to start at a location within the crater resulted in the Truck being correctly oriented on the surface of the terrain within the crater. This result meets the goal that the protocol support entities which join the simulation late, possibly after the terrain has been modified by the terrain server.

Scenario III is not a practical one yet it illustrated the effectiveness of the Checksum Tree Protocol. Restarting the dynamic terrain server mid-course in a simulation results in the terrain database reverting from its mutated form to its initial state. From the vantage point of the Stealth, craters disappear, and vehicles in craters pop up to the original surface of the terrain. This demonstrates that all clients successfully become updated via the Checksum Tree Protocol.

6.6 Virtual Server Testing Scenario

The Virtual Server implemented for this thesis is a Level 1 fault tolerant server (see Chapter 4) consisting of a master server and a single slave server. Testing the Virtual Server involves stopping the master server process after the dynamic terrain database has been

modified, and ensuring that the slave server takes over as the master retaining any mutations to the terrain database.

A master server will be started on one workstation and the slave server on another. PDUGEN is used to issue a DetonationPDU causing the master server to modify the terrain database. Like client simulations, the slave server uses the Checksum Tree Protocol to maintain a consistent version of the terrain database with the master server. A control-C will be issued to the master server process causing it to terminate. The slave server is designed to take over as the master if it does not receive a heartbeat message from the master within two heartbeat intervals. Diagnostic "printf" calls have been inserted into the server code to indicate changes in master/slave status. After stopping the master server process, the Stealth is brought on-line in a location proximate to the point of detonation. If the slave server correctly takes over as the master and has a consistent version of the terrain database with the former master, the new master server will update the Stealth's terrain database via the Checksum Tree Protocol. This is visualized by the Stealth out-the-window view.

6.6.1 Virtual Server Testing Results

Within two heartbeat intervals after stopping the master server process, the slave server printed the diagnostic indicating that it was switching to master mode. The Stealth was then brought on-line. Upon the receipt of the first heartbeat message from the new master, the Stealth initiated the Checksum Tree resolution process and was updated by the new master server. This visually manifested itself as a crater appearing in the field-of-view of the Stealth immediately upon initialization.

7 Conclusions and Suggestions for Further Research

The Checksum Tree Protocol has been shown to be a viable approach to solving the reliable dynamic terrain update problem. This constitutes a significant contribution to the DIS community. When a standard terrain representation scheme is defined which meets the needs of image generators, man-in-the-loop simulators, and computer generated forces, the Checksum Tree will be available to ensure reliable and efficient updates to distributed dynamic terrain databases. The result will be an improved realism for simulations which will benefit training, virtual prototyping, and entertainment applications.

Although much research has been done in the area of reliable broadcast, most of the proposed approaches rely on receiver-generated acknowledgments. The large number of potential receivers in a distributed interactive simulation coupled with the real-time nature of such simulations makes the existing reliable broadcast protocols impractical for supporting the needs of distributed dynamic terrain. The Checksum Tree is not a general protocol which supports reliable broadcast of arbitrary data. Instead, the Checksum Tree is a problem domain specific approach which exploits knowledge of the structure of the data being broadcast. Advantages to this approach are summarized below:

1. Clients receive terrain updates instantly and simultaneously at the point of terrain mutation via a broadcast PDU.
2. The impact on network bandwidth is minimized because most clients receive the information from a single broadcast.

3. Servers need not know the identity and number of clients because no "check - in" protocols are used.
4. The protocol is stateless, simplifying the client-side application code. This reduces coding errors and makes the protocol more "sellable" to simulation developers.
5. Requires minimal modification to the existing DIS standards.
6. The first client which elicits a retransmittal by the server benefits slower clients which missed the same terrain update message.

The Virtual Sever successfully created the abstraction of a reliable terrain serving entity on the network. A redundant server is required to remain consistent with the DIS philosophy of no reliance on a central computer for event scheduling for conflict resolution. The Virtual Server abstraction simplifies the client-side applications because the clients appear to be communicating with a single reliable server. Terrain database consistency among the members of the Virtual Server is promoted when the Virtual Server is used in conjunction with the Checksum Tree Protocol.

7.1 Checksum Tree Technical Considerations

The interval between heartbeat messages from the terrain server must be chosen to be sufficiently large as to allow for missed terrain update messages to be completely resolved between the client and the server. This parameter also determines the latency between when a client realizes that it has missed a terrain update message, and the time in which the resolution process is initiated. For this reason, this parameter should not be chosen to be too large. Studies should be conducted to determine the optimal value for this parameter

considering the nature of the simulation exercise, number of simulators and entities participating in the simulation, and the simulator and network hardware available.

Throughout the course of this research, it has been assumed that the terrain database is simply a triangular mesh represented by a grid of elevation samples, ignoring any cultural or feature data. Terrain representation schemes must be devised which allow network transmission of terrain feature data as well. This implementation also assumed that updates to a patch of terrain fit into a single PDU. Dynamic mutations to the terrain database (e.g. craters, berms, holes) require a much higher resolution representation than the 125 meter elevation grid used by most SIMNET based terrain databases. It may be determined in the future that a sufficient fidelity terrain representation scheme would require terrain patch updates to span multiple PDUs. This would require extending the Checksum Tree Protocol to deal with this complication. The tradeoff between smaller patch sizes versus multiple PDUs for a terrain patch update should be studied.

7.2 Virtual Server Technical Considerations

Considering the current state of network technology, it is impractical to pursue Level II fault tolerance for redundant servers connected by a wide-area network. The delays which may accumulate as messages pass through gateways prevents a slave server from reliably determining that a master is off-line within a reasonable time window. As networking hardware improves over time or as new technologies evolve, this constraint may disappear in the future.

The dynamic terrain server demonstrated in this thesis was a simplistic one designed solely for the purposes of evaluating the Checksum Tree Protocol and the Virtual Server concept. This simple server created craters in the terrain as a result of detonations sufficiently close to the terrain surface. A more realistic server would take the warhead type and soil attributes into account when assessing the impact of a detonation on terrain.

Furthermore, other terrain mutating events such as a vehicle colliding with the terrain should be handled as well.

An even more advanced terrain server would allow for entity initiated modifications to the terrain. As an example, a bulldozer may wish to create a berm. Such a scenario would likely require a distributed locking protocol⁴⁴ to handle the case of multiple entities trying to simultaneously modify the same patch of terrain.

Bibliography

- ¹ "A Virtual Cockpit for a Distributed Interactive Simulation", W.D. McCarty, *IEEE Computer Graphics and Applications*, Vol 14 Page 49-54, 1994
- ² "Advanced Weapons Team Training Technology", R. McCormack, A. Marshall, R. Wolff, J. Horey, E. Purvis", *Proceedings from the 15th Conference on Interservice/Industry Training Systems and Education*, Pages 327-335, 1993
- ³ *The SIMNET Network and Protocols*, A. Pope, BBN Report No. 7627, 1991
- ⁴ "The New Technology of Large Scale Simulator Networking: Implications for Mastering the Art of Warfighting", J.A. Thorpe, *Proceedings from the 9th Conference on Interservice/Industry Training Systems and Education*, Page 492-501, 1987
- ⁵ *IEEE Standard for Information Technology - Protocols for Distributed Interactive Simulation Applications*, IEEE Press, New York, May 12, 1993
- ⁶ "Modeling and Simulation of Communications", V.M. Bettencourt, *IEEE MILCOM '92 - Communications - Fusing Command, Control, and Intelligence*, Vol I Page 90-93, 1992
- ⁷ "Addressing the Need for Software Tools to Meet Exercise Support and Feedback Requirements", R.M. Rybacki, L.D. Snyder, G.L. Naville, *Proceedings from the 9th Workshop on Standards for the Interoperability of Defense Simulators*, Page A21-A30, 1993
- ⁸ "Opportunities for Intelligent Vehicle Highway Systems - Applications of Distributed Interactive Simulation", W.G. Smith, D.A. Whitney, R.M. Rybacki, *Proceedings from the 1994 Summer Computer Simulation Conference*, Page 191-200, 1994
- ⁹ "Distributed Interactive Simulation for Emergency Response Training", P.L. Versteegen, J. Rubenstein, *Proceedings from the 1994 Summer Simulation Conference*, Page 117-124, 1994
- ¹⁰ "Distributed Air Traffic Management Simulation: The Next Application Domain for DIS", R. Stevens, W. Neal, *Proceedings from the 1994 Summer Simulation Conference*, Page 279-291, 1994
- ¹¹ "The Software Architecture for Scenario Control in the Iowa Driving Simulator", J.K. Kearney, J. Cremer, *Proceedings from The Fourth Conference on Computer Generated Forces and Behavioral Representation*, Page 373-381, 1994
- ¹² "Computer Generated Images from Digital Grid Databases", R.A. Heartz, *IEEE SOUTHEASTCON '89 Proceedings, Vol II* Page 884-887, 1989
- ¹³ "Environmental Extensions to Modular SemiAutomated Forces", R.L. Schaffer, *Proceedings from The Fourth Conference on Computer Generated Forces and Behavioral Representation*, Page 17-23, 1994
- ¹⁴ "Suitability of the Standard Simulator Database Interchange Format for Representation of Terrain for Computer Generated Forces", T. Stanzione, *Proceedings from The Fourth Conference on Computer Generated Forces and Behavioral Representation*, Page 231-238, 1994
- ¹⁵ *Dynamic Terrain PDU Analysis via Use Cases*, Curtis Lisle, UCF IST Visual Systems Lab Technical Report, 1994
- ¹⁶ "Dynamic Environment Simulation with DIS Technology", M. Kirby, C. Lisle, M. Altman, M. Sator, *Proceedings from the 16th Conference on Interservice/Industry Training Systems and Education*, Page 418, 1994
- ¹⁷ "Architectures for Dynamic Terrain and Dynamic Environments in Distributed Interactive Simulation", C. Lisle, M. Altman, M. Kilby, M. Sartor, *Proceedings from The 10th Workshop on Standards for the Interoperability of Defense Simulators*, Page A50-A61, 1994
- ¹⁸ "An Object-Oriented Environmental Server for DIS", W.H. Horan, M.J. Smith, C.R. Lisle, M. Altman, *Proceedings from the 9th Workshop on Standards for the Interoperability of Defense Simulators*, Pages A95-A113, 1993
- ¹⁹ *UNIX Network Programming*, W.R. Stevens, Prentice-Hall, 1990
- ²⁰ "Dynamic Multicast on Asynchronous Transfer Mode for Distributed Interactive Simulation", T.L. Gehl, *Proceedings from the 16th Conference on Interservice/Industry Training Systems and Education*, Page 4.3, 1994
- ²¹ "Reviewing the Battle at the Alamo", G.C. Mak-Cheng, K. Doris, *Proceedings from the 15th Conference on Interservice/Industry Training Systems and Education*, Page 612-618, 1993
- ²² "The Performance Assessment of the Dead Reckoning Algorithms in DIS", K.C. Lin, D. Schab, *Proceedings from The 1994 Summer Computer Simulation Conference*, Pages 305-314, 1994
- ²³ *Computer Networks*, A.S. Tanenbaum, Prentice-Hall, 1988
- ²⁴ "Dynamic Terrain", J.M. Moshell, B. Blau, X. Li, C. Lisle, *Simulation Magazine January 1994 Edition*, Page 43-58, 1994
- ²⁵ *Computer Graphics*, Foley, Van Damm, Addison-Wesley, 1991
- ²⁶ *The Design and Implementation of the 4.3 BSD UNIX Operating System*, Leffler, McKusick, Karels, Quarterman, Addison-Wesley, 1989
- ²⁷ "Background Clutter Simulation", W.G. Smith, R.M. Rybacki, T. Stanzione, *Proceedings from The Fourth Conference on Computer Generated Forces and Behavioral Representation*, Page 385-394, 1994
- ²⁸ *Strawman Distributed Interactive Simulation Architecture Description Document Volume I*, Loral Systems Company Technical Report, 1992
- ²⁹ *Local Area Networks*, G.E. Keiser, McGraw-Hill, 1989
- ³⁰ "An Efficient Reliable Broadcast Protocol", M.F. Kaashoek, A.S. Tanenbaum, S.F. Mummel, H.E. Bal, *ACM Operating Systems Review Journal*, Page 5-17, 1989
- ³¹ "The Multidriver: A Reliable Multicast Service using the Xpress Transfer Protocol", B.J. Demmpsey, J.C.Fenton, A.C. Weaver, *Proceedings from the 15th IEEE Conference on Local Computer Networks*, Page 351-358, 1990
- ³² "Reliable Broadcast Protocols", J. Chang, N.F. Maxemchuk, *ACM Transactions on Computer Science*, Pages 251-273, 1984

-
- ³³ "A Token-Based Protocol for Reliable, Ordered Multicast Communication", B. Rajagopalan, P. McKinley, *Proceedings of Eight Symposium on Reliable Distributed Systems*, Pages 114-122, 1989
- ³⁴ *Distributed Systems Concepts and Design*, G. Coulouris, J. Dollimore, T. Kindberg, Addison-Wesley, 1994
- ³⁵ *Distributed Interactive Simulation Operational Concept*, UCF Institute for Simulation and Training Technical Report, 1993
- ³⁶ "Distributed Fault-Tolerant Real-Time Systems: The Mars Approach", H. Kopetz, A. Damm, C. Koza, M. Mulazzani, W. Schwabl, C. Senft, R. Zainlinger, *IEEE Micro*, Volume 9, Page 25-40, 1989
- ³⁷ "Leader Election in Uniform Rings", Shing-Tssan Huang, *ACM Transactions on Programming Languages and Systems*, Vol. 15 Page 563-573, 1993
- ³⁸ *Data Communications - A User's Guide*, K. Sherman, Reston Publishing, 1985
- ³⁹ "Performance Evaluation of a Real-Time Fault Tolerant Distributed System", L. Alger, J. Lala, *Proceedings of the Twenty-Third Annual Hawaii International Conference on System Sciences*, Page 278-287, 1990
- ⁴⁰ *Cognitive Ergonomics: Contributions from Experimental Psychology*, G.C. Van Der Veer, S. Bagnara, G. Kempen, North-Holland, 1992
- ⁴¹ *Object Oriented Design with Applications*, G. Booch, Benjamin/Cummings 1991
- ⁴² *Fundamentals of Data Structures*, E. Horowitz, S. Sahni, Computer Science Press, 1982
- ⁴³ *Operating Systems - Design and Implementation*, A.S. Tanenbaum, Prentice-Hall, 1987
- ⁴⁴ *Operating Systems Advanced Concepts*, M. Maekawa, R. Oldehoeft, Benjamin/Cummings, 1987

VITA

Richard Michael Rybacki was born in South Bend, Indiana on June 19, 1965, the son of David Lee Rybacki and Emily Jean Rybacki. After completing his work at John Marshall High School, San Antonio, Texas, in 1983, he entered the University of Texas at San Antonio. In March of 1986, he joined the United States Air Force and completed his Bachelor of Science in Computer Science concurrent with his military service in May, 1990. In September, 1990, he entered the graduate program in Computer Science at the University of Texas at San Antonio. In January 1991, he was honorably discharged from the Air Force and has since been employed by TASC, Inc., as a Software Engineer.

Permanent address: 7171 Wurzbach Apt. 201
 San Antonio, Texas 78240

This thesis was typed by the author.