The NAVL Distributed Virtual Reality System*

Martine Wedlake, Kin F. Li, and Fayez El Guibaly

Department of Electrical and Computer Engineering,
University of Victoria
Box 3055
Victoria, BC CANADA V8W 3P6
{martine, kinli, fayez}@engr.uvic.ca

Abstract. Current distributed virtual reality systems suffer from many inherent difficulties: inadequate network architecture, poor object distribution and coherency, poor system resource management, and poor overall performance. This paper outlines the NAVL DVR system, designed to overcome these problems with a distributed client/server network architecture and master/slave autonomous object distribution mechanism using ForceLet simulation.

1 Introduction

Distributed virtual reality (DVR) is a form of distributed multimedia (DM). Unlike many DM applications, which have minimal interaction with the participants, DVR systems have a very high degree of interaction. While there are some DM applications that exhibit degrees of interaction, they are often limited by poor quality voice and slow refresh video (e.g., video conferencing).

DVR participants need quick access to virtual objects[1] containing multimedia content (e.g., computer graphics, sound and text). A major challenge for DVR system developers is to find ways to support highly interactive environments with limited system resources.

Distributed VR systems can be viewed as a collection of five service layers, listed from top to bottom:

1. The **Application Layer** represents the DVR applications such as virtual conferencing.
2. The **Language/Application Programming Interface (API) Layer** supports DVR Applications through a structured interface between applications and the PS Layer.
3. The **Programming System (PS) Layer** provides the infrastructure for managing DVR environments.

* This work was supported by the Natural Sciences and Engineering Research Council of Canada.

[1] In this paper, the term virtual object refers to the object in the virtual world. This removes confusion arising due to object oriented programming.

4. The **Operating System (OS) Layer** contains machine services such as CPU scheduling and memory allocation.
5. The **Protocol Layer** defines the structure and mechanisms of communication between remote hosts in a DVR system.

1.1 Case Studies

This section outlines a summary of some of the more prominent DVR systems found in the literature and how they fit into the DVR layer structure; the DVR layers involved are listed parenthetically (notice how many DVR systems are focused on a small subset of the DVR layers).

VRML (2, 5)[11] was created to provide a standard framework for developing VR languages over the Internet. It provides a language element to describe scenes, and a protocol for sending the scene descriptions over the Internet (similar to HTTP).

DIVE (2, 3)[1] uses a fully distributed shared database model. Applications change node-local copies of virtual objects, which are replicated by the DIVE infrastructure to all other participants. Mutual exclusion operators are provided to serialize simultaneous updates.

Alpha World (1)[2] is an application available over the Internet that allows multiple people to socialize in a three-dimensional setting - essentially a DVR chat program. The architecture is client/server with special gateways that one can travel through to switch servers.

MR Toolkit with Peers Package (2, 3)[9] is a collection of libraries that can be used to build small DVR applications (less than five participants) using a fully connected network model.

DIS (5)[4] describes a protocol for managing warfare simulators over a wide range of communication media. To reduce network load, DIS uses a dead reckoning protocol. Each simulator maintains two states: the internal (e.g., real) state, and the approximate (e.g., dead reckoning) state. Only the dead reckoning state is transmitted to receiving applications. When the dead reckoning state diverges from the internal state, the dead reckoning state is updated to match the internal state. Messages are only transmitted to receiving applications when the dead reckoning state changes.

1.2 Shortcomings with Existing DVR Systems

Many of the existing DVR systems exhibit shortcomings that limit their potential in practical DVR applications. The following four areas needing improvement have been identified:

1. **Network Efficiency**: This involves managing both bandwidth and latency of the network. These factors can be controlled by the design of the network architecture and efficient use of the available resources.
2. **Object Distribution and Coherency Models**: Proper management of virtual objects in a DVR system is critical to ensure good scaling performance as the system grows.

3. **Inadequate System Resource Management**: As with OS developers, DVR developers need to control many diverse kinds of resources (e.g., network, computation, graphics, and memory). Indeed, many DVR developers leverage OS services directly to manage system resources. However, while OS developers are typically interested in providing fairness[10], DVR system developers are concerned with providing immersion and interaction to the participant.
4. **Overall Performance**: The authors define a high performance DVR system as high synchronization fidelity, low environment lag, low display lag, large number of participants, and high world complexity.

1.3 Description and Motivation of NAVL

Many current DVR systems tightly couple the application layer through to the programming system layer. Often to the extent that the entire DVR system must be re-compiled in order to extend the existing application. The Newtonian Architecture for Virtual Landscapes (NAVL), takes a different approach. Instead of bonding the application to the operating environment, NAVL tries to separate the two. Application Programming Interfaces (APIs) provide a general purpose environment to support DVR applications. The NAVL architecture incorporates the following four elements:

1. **Autonomous virtual objects**: allows an encapsulation of the virtual object that may be moved as needed to balance the CPU and networking loads of the DVR system.
2. **Distributed client/server network architecture**: reduces bottlenecks within the network by spreading the bandwidth throughout the network, while simultaneously reducing the load to the network as compared with a peer-to-peer architecture.
3. **Master/slave virtual object distribution and coherency model**: splits the virtual object into a single master object and several slave objects. The master object is responsible for the state of the object while the slave objects represent the local manifestation of the object. The slave objects execute high-order commands called ForceLets that describe a path through space for the object to follow.
4. **Local simulation of virtual objects by slave objects**: reduces network bandwidth requirements by executing much of the virtual objects actions local to the user rather than at a remote node.

Unlike traditional DVR systems, NAVL supports both applications and autonomous virtual objects. This has the effect of splitting the DVR layers into dual stacks (see Fig. 1) for the applications and the virtual objects.

An application in the NAVL model requires both the application code and one or more virtual objects (represented as master objects). Applications communicate their requests to the programming system via an API, while the autonomous virtual object communicates via an object programming interface (OPI). In general, the virtual objects may be regarded as "toolkits" for application developers.

(1)	Application			Master Object
(2)	Rendering API	NAVL API		NAVL OPI
(3)	Renderer	Programming System		
(4)	Window System	Operating System		
(5)	Window System Protocols	Protocols		

Fig. 1. NAVL DVR layers. Notice how the DVR layers are split to accommodate applications and master objects

2 NAVL DVR Part I: Design Architecture

NAVL was designed as an answer to the observed problems in DVR systems. Many of the problems are handled by NAVL by the use of the master/slave coherency model. Figure 2 shows the high-level design architecture with the virtual object represented as a master object and two slave objects.

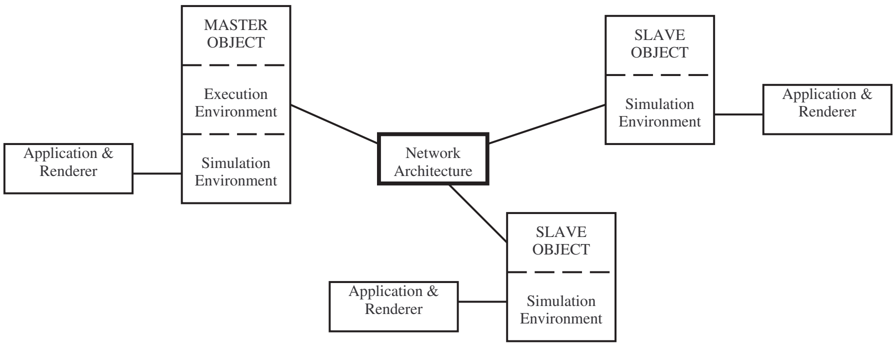


Fig. 2. Design architecture elements. The execution environment and simulation environment represent the set of services provided by the NAVL programming system

The *master objects* are responsible for maintaining the virtual object's state, and *slave objects* replicate the virtual object throughout all the NAVL hosts[2]. Unlike client/server paradigms, applications make requests of the objects themselves. When the master object resides on the same host, it is invoked directly. When the master object resides on a remote host, the application communicates to the local slave object. The slave will forward the request to the master object automatically.

The NAVL DVR system supports distributed interactive multimedia applications (e.g., virtual reality), which typically have high demands for 3D computer

[2] NAVL hosts are machine nodes that participate in the NAVL DVR system.

graphics. NAVL encapsulates these services into a rendering engine library linked directly in the application.

The rendering engine is linked in the application (instead of being provided as a standard NAVL service) because most applications are likely to use more than a simple 3D viewspace. The use of window system primitives (e.g., menus, buttons, and dialog boxes) are left up to the application developer. Placing the renderer in the application allows the greatest degree of flexibility for the application designer.

2.1 Network Architecture

Network architectures describe how information will move throughout the DVR system, three example architectures are shown in Fig. 3.

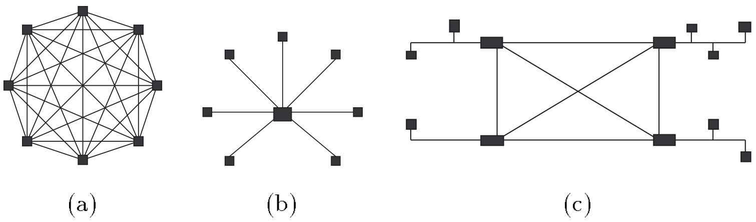


Fig. 3. Network architectures: (a) peer-to-peer, (b) client/server and (c) distributed client/server

Peer-to-Peer network architectures have interconnections between every host, distributing the network load evenly throughout the network, at the expense of more message traffic.

Client/Server network architectures have a special node designated the server that services requests from a collection of client nodes. The client/server concentrates the network into the server node, thereby reducing the number of messages sent over the network as a whole, but increases the load at the central point. The server may opt to buffer messages for a specified duration in order to reduce network bandwidth (at the expense of latency), this is deemed the buffered client/server network architecture.

Distributed Client/Server is a combination of the client/server and peer-to-peer architectures. The network is divided into regions, several local area networks (LANs) and the single wide area network (WAN). The LANs are broadcast driven, while the WANs are peer-to-peer. The larger boxes in Fig. 3(c) are special message servers called message managers, very similar to those in WAVES and HIDRA[7, 8]. Message managers serve four functions:

1. Act as a gateway between the peer-to-peer WAN and the broadcast LAN.
2. Perform directory services (e.g., host and object lookup).

3. Filter messages coming into and leaving the LAN to reduce bandwidth over the WAN.
4. Regulate the quality of service (QoS) services of the WAN, if any.

Analysis of network capacity has shown that the distributed client/server network architecture fairs very well against the peer-to-peer, and simple client server network architectures[13]. For this reason, NAVL uses the distributed client/server network architecture.

2.2 Object Distribution and Coherency Architectures

Distributed virtual reality applications need assurances that actions taken by different participants can and will have an impact on the other participants. To meet this need, an efficient distribution and coherency model is necessary. The literature has shown many examples of distribution and coherency models, for example:

Dead Reckoning is used by the DIS standard[4] to reduce network bandwidth. However, the use of two virtual world states (simulated and real) and lack of extensible dead reckoning functions limit the viability of this method.

Centralized Database maintains the virtual objects state information in a single server. This is used by the client/server systems such as Alpha World[2]. The lack of distribution of state data causes bottlenecks within the network fabric, which are only addressed by deploying larger and faster machines.

Distributed Database Locking enforces consistency by restricting access to the database itself. The DIVE[1] DVR system locks the database with mutual exclusions (MUTEX). Locking provides a good mechanism for ensuring that transactions are carried out sequentially, however using a locking mechanism for distributed applications involves a lot of network traffic resulting in long time lags. Typical distributed locking schemes[3] require messages to be sent repeatedly to all hosts; making it nearly impossible at current network latencies to have real-time interaction.

Autonomous Objects with Master/Slave Replication distributes a virtual object throughout the network, designating one copy as the master object and the rest as the slave objects. The master object decides what actions to take and is responsible for replicating that task to all the slave objects, using a high-level simulation protocol. The NAVL DVR system[13, 14] and HIDRA DVR system[8] use autonomous objects.

2.3 Autonomous Objects with Master/Slave Replication

Master objects are responsible for executing special behavior routines upon receiving an event, resulting in either changing the virtual objects state or issuing an event of its own. Changes to the virtual objects state affects the visual manifestation of the object (e.g., color or shape) or location. Figure 4 shows the block diagram for NAVL virtual objects. The master and slave components are shaded for clarity.

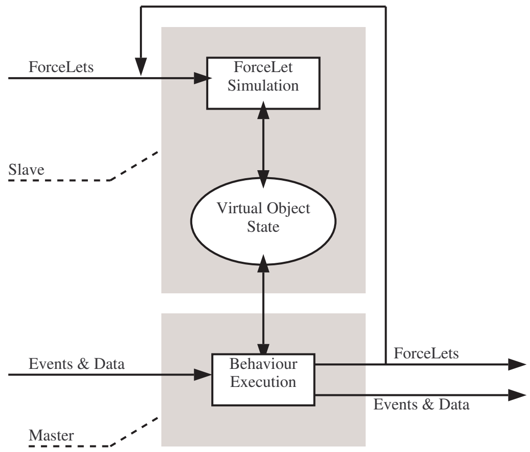


Fig. 4. Block diagram of virtual object

A special simulation environment is used when moving the virtual object through space. The simulation environment must be sufficiently complex to allow for a large range of actions, must be predictive, and must use little network bandwidth resources.

The current model for the simulation environment is based on a modification of Newtonian mechanics. ForceLets are network messages sent from the master object to the slave objects to define the path virtual objects take through space and time. ForceLets contains the following:

- Recipient and Sender Object Identifier
- Type of ForceLet
- Parameters for the ForceLet

The ForceLet notation is $F_D(A, \tau, T; t)$ where D specifies the domain for the definition (f for force, v for velocity, or d for displacement), and the parameters are: A for amplitude, τ for the start time, and T for the duration. The point in time is specified by t and is synchronized to all hosts using the network time protocol.

The authors have initially defined three ForceLets: the $\frac{1}{2}$Cosine, Sine, and Step. As research progresses, it is expected that additional ForceLet types will become identified and added to the family of ForceLets. Naturally, the list of ForceLet parameters may increase during this process as well.

The **Step ForceLet** is defined as follows:

$$F_f(A, \tau, T; t) = \begin{cases} 0 & \text{for } t - \tau \leq 0 \\ A & \text{for } 0 \leq t - \tau < \frac{1}{2}T \\ -A & \text{for } \frac{1}{2}T \leq t - \tau < T \\ 0 & \text{for } T \leq t - \tau \end{cases} \tag{1}$$

This models the case when the acceleration of the virtual object is constant until halfway through the journey, at which point it decelerates at a constant rate until zero velocity.

The $\frac{1}{2}$**Cosine ForceLet** uses the first half of the cosine wave to smooth out the transition of acceleration to deceleration as is seen with the step ForceLet. It is modeled as:

$$F_f(A, \tau, T; t) = \begin{cases} 0 & \text{for } t - \tau \leq 0 \\ mA \cos\left(\pi \frac{t-\tau}{T}\right) & \text{for } 0 \leq t - \tau < T \\ 0 & \text{for } T \leq t - \tau \end{cases} \tag{2}$$

The Sine ForceLet uses the entire sine wave to smooth out the start and end points. The acceleration and deceleration is applied gently. This allows a smooth transition from one ForceLet to another, and is especially useful when combining ForceLets. It is modeled as:

$$F_f(A, \tau, T; t) = \begin{cases} 0 & \text{for } t - \tau \leq 0 \\ mA \sin\left(2\pi \frac{t-\tau}{T}\right) & \text{for } 0 \leq t - \tau < T \\ 0 & \text{for } T \leq t - \tau \end{cases} \tag{3}$$

The Step ForceLet uses the least computation, while the Sine ForceLet uses the most[14]. Conversely, the step ForceLet is the least smooth when combining ForceLets, while the sine ForceLet is the smoothest. The $\frac{1}{2}$Cosine ForceLet fits between the two in complexity and smoothness.

2.4 The Application and Rendering Components

The application provides a semantic structure to the virtual objects. It is responsible for creating, manipulating, displaying and destroying virtual objects. Rendering is the process of displaying the virtual object on the video screen or through VR helmet. Since the display is tightly integrated with the application, it is shown together in Fig. 2.

3 NAVL DVR Part II: Implementation Architecture

Figure 1 shows three distinct parts of the NAVL system: (1) the application code *(top left of the figure)*, (2) the master object *(top right of the figure)*, and (3) the NAVL programming environment *(middle of the figure - the programming system and renderer layers)*.

These parts of the DVR model are implemented with distinct subsystems as seen in Fig. 5(a). This figure shows the implementation architecture for a single node within the NAVL system (Fig. 5(b) is discussed in the next section). The shaded regions show the separation of the process groups. The upper box represents the NAVL system, and the lower box represents the application process.

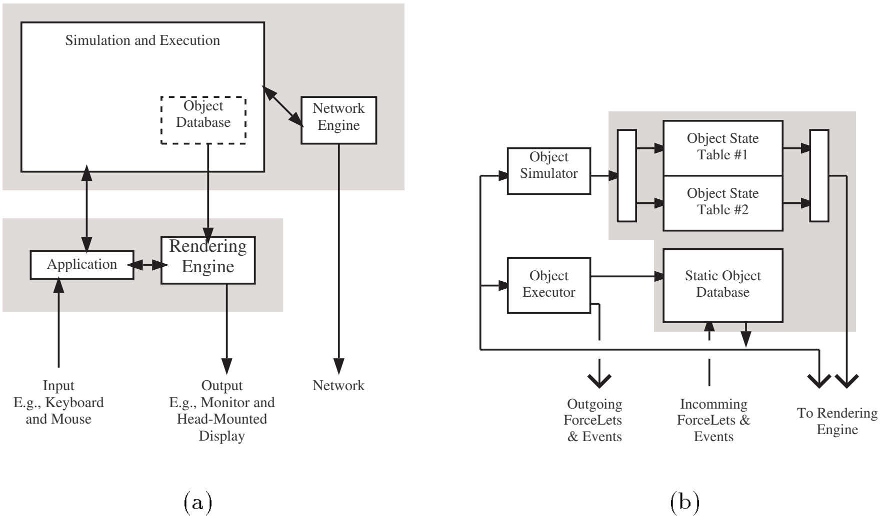


Fig. 5. Block diagram of (a) implementation architecture, and (b) Simulation and Execution Architecture. The shaded regions in (a) represent the groupings of user processes and NAVL processes, and in (b) represent shared memory

3.1 Simulation and Execution Engine

The largest and most critical element of the implementation architecture is the Simulation and Execution Engine (SEA). Its purpose is to replicate virtual objects according to the master/slave paradigm. The block diagram of the SEA is shown in Fig. 5(b). This figure is the dual to the virtual object diagram (Fig. 4), which shows object execution from the system perspective instead of the virtual object perspective.

The object simulator continually applies ForceLets to slave objects to obtain the object state table. The object state table represents the point-in-time state of all the slave objects in the NAVL host. There are two object state tables to store the previous frame while constructing the next frame. The rendering engine (e.g., application) will always retrieve data from the previous frame, not the frame being constructed. This preserves integrity during the rendering process and allows them to execute in parallel without blocking on a MUTEX.

The object executor executes behavior execution routines contained in master objects upon receipt of events (not shown). The result of the behavior execution can be a series of one or more issued ForceLets, or other state changes (e.g., shape, color or orientation).

The shaded area of Fig. 5(b) show the data structures stored in shared memory for simultaneous access from the simulation and execution engine and the rendering engine. The use of shared memory also enables future use of symmetric multi-processor (SMP) architectures whereby multiple object executors can exist on different physical processors communicating via shared memory.

Virtual Object Execution The virtual object executor executes events found in the event queue. Events contain:

- The sending and receiving object identifiers.
- The type of event (e.g., messages, timers and collisions). Message events are created and issued by applications and virtual objects as a means of communication. Timer events are set by the master object to wake itself up at a specified time. Collision events are issued to virtual objects involved in a collision when detected. The collision behavior routines in the virtual objects involved are responsible for coordinating a course of action.
- The data payload (arguments) for the event.

The virtual object executor runs the master object's behavior routine. During execution time, the behavior routine can issue ForceLets, change the virtual objects visual manifestation, or send events to itself or other virtual objects. However, the behavior execution routine must return within a specified time. This is very much like a soft real-time system. Long duration tasks can be scheduled by the user of a repeatable timer that periodically wakes up the master object. The soft real-time scheduling paradigm builds the foundation for quality of service (QoS) mechanism that can be embedded into the system at a later time.

Virtual Object Simulation The object simulator calculates the position of virtual objects based on the list of ForceLets, and the global simulation time. The simulator runs through the list of slave objects at the base simulation rate (usually about 10–20 Hz), placing the result into the active object state table.

The base simulation rate establishes the environment lag for the NAVL system, and should therefore be kept constant whenever possible. The base simulation rate is currently hard coded into the NAVL DVR prototype, however there is no reason it can not be made to adjust to the current hardware available on the host. The simulation environment decouples the NAVL hosts, allowing them to have different base simulation rates.

SEA Support Infrastructure The SEA supports the application and master objects through interfaces called the application programming interface (API) and the object programming interface (OPI).

The API uses shared memory to communicate between the NAVL system and the application. This is dramatically faster than other mechanisms such as named pipes or sockets, however it does require the application to execute on the same node as a NAVL host. Since the NAVL model extends deeply into the DVR layers, it is expected that the application will live on the same node.

Master object behavior routines are loaded into NAVL using shared library load operations, therefore the OPI is invoked through direct function calls. This allows for a more efficient interface than using the available shared memory, however there is an increased risk to the NAVL system's integrity. Security has not been addressed at this time.

3.2 Rendering Engine

The rendering engine performs two specific tasks: (1) draws scenes representing the visual manifestation of the virtual objects, and (2) detects collisions between virtual objects.

Objects are usually represented using either surface or solid modeling[5]. NAVL uses solid modeling of known shapes (e.g., cube, sphere, cone, and torus) for simplicity. While not as visually pleasing as can be obtained via surface modeling, it can be easier to use for defining shapes. The OpenGL graphics library has been chosen to provide the low level graphics rendering. This reduces the burden of the application developer from writing the rendering code itself.

To provide complex shapes, the NAVL prototype allows for shapes to be arranged in a hierarchy. The children objects live within the coordinate system of the parent. This arrangement allows for quick and easy translations and rotations of large components of the composite virtual object. The origin shape does not have any visual components, and is used to provide a reference object for coordinate transformations.

Because the rendering engine models the shape of virtual objects, it is the prime location for detecting collisions. Detecting collisions between virtual objects in a virtual environment requires an efficient algorithm for recognizing collisions between objects of arbitrary shapes and also an efficient algorithm for searching through the list of virtual objects looking for possible collisions.

The first part, recognizing collisions, depends on the shape model chosen. For example, solid modeling is typically easy to detect collisions for since they are often based on conics or other easily managed objects. Surface modeling, however causes difficulties due to the large number of polygons and arbitrary shape.

The second part, organizing the search space, is often handled in one of the following ways:

1. **Nested Loop** simply loops through all combinations of object pairs to determine if a collision occurred. Naturally, this mechanism is the most costly in terms of CPU.
2. **Locale or Segmented** divides the virtual world into regions of nearby space. The advantage is twofold: first, locales that do not touch each other can be searched in parallel; and second, the number of pairs is reduced since virtual objects that are not in the same or touching locales can not collide.
3. **Predictive** examines the virtual objects' paths through space and determines the earliest time a collision could occur[6]. When the virtual environment uses predictable paths, this can be determined fairly easily. When the virtual environment does not use predictable paths, then other assumptions must be made (e.g., fastest closure rate) to determine earliest collision time.

To detect collisions, the NAVL prototype recognizes collisions on the solid models with a simple nested loop search algorithm. It is recognized that the simulation environment within NAVL should provide a good platform for predictable collision detection. The authors are currently looking into this approach.

3.3 Network Engine

The network engine communicates ForceLets and events over the LAN using a broadcast datagram. All NAVL hosts on the LAN listen for ForceLets and events. If the ForceLet or event refers to a virtual object in the host, it handles the request immediately (via the same broadcast facility).

The message manager can filter incoming and outgoing messages to reduce network load on the WAN. In addition, it can request QoS levels commensurate with the expected load and importance of the traffic.

The NAVL prototype implements an imperative-based network protocol for communicating between the NAVL hosts on the LAN. The message manager's role has been reduced to a name service for simplicity and lack of resources for evaluating larger WAN-based implementations.

The messages in the protocol are phrased as commands to execute on the remote host, broken into five categories:

1. **Object Management** messages govern the creation, and destruction of virtual objects. Creating a slave object is called "attaching" to the master object. This emphasizes the master/slave relationship. These messages are sent to the host managing the master object.
2. **Object State Change** messages are sent by the master object whenever the virtual object changes its state. For example, ForceLets, and changing shape or orientation, require an object state change message to be broadcast.
3. **Name Service** messages enable applications to "publish" master objects so that remote hosts can attach to them. These messages are only sent between the message manager and the requesting host or application.
4. **Host Initialization** messages inform the new NAVL host of the current status of the NAVL virtual world. These messages are sent between the message manager and the new NAVL host.
5. **Simple** messages allow communication between applications and master objects. These messages are sent directly to the host with the master object.

4 NAVL DVR Part III: Evaluation

The NAVL DVR system is evaluated based on the following components: the distributed client/server network architecture, the ForceLet-based coherency model, and the OpenGL-based rendering engine with collision detection.

4.1 Network Architecture

The NAVL network architecture tries to reduce the network load by combining the peer-to-peer and client/server network paradigms. An evaluation of the capacity of the network has been made in [13]. The analysis determines the total number of packets transmitted in the network, and the peak number of packets transmitted through a node when varying number of messages (distributed throughout the network) are transmitted to all other hosts.

This evaluation concludes that the NAVL network architecture is the best compromise for minimizing both total network traffic (best) and peak network traffic (second best). To show these results visually, Fig. 6 shows the total number of messages and the peak number of messages for varying number of messages inserted into the system (we assume that no host will send more than one message). The parameters for these graphs are as follows: 500 hosts, 10 message managers, and an even distribution of message source locations.

Latency of the NAVL network architecture, however, will be slightly longer than competing architectures due to the additional hops imposed by the message managers. However, the authors feel that the distribution of the network load, QoS services, and message filtering more than compensate for the additional hops. In addition, it is expected that the network architecture will be deployed such that applications accessing common objects will be contained in the same LAN. For many cases, this will remove the hops imposed by the message managers entirely.

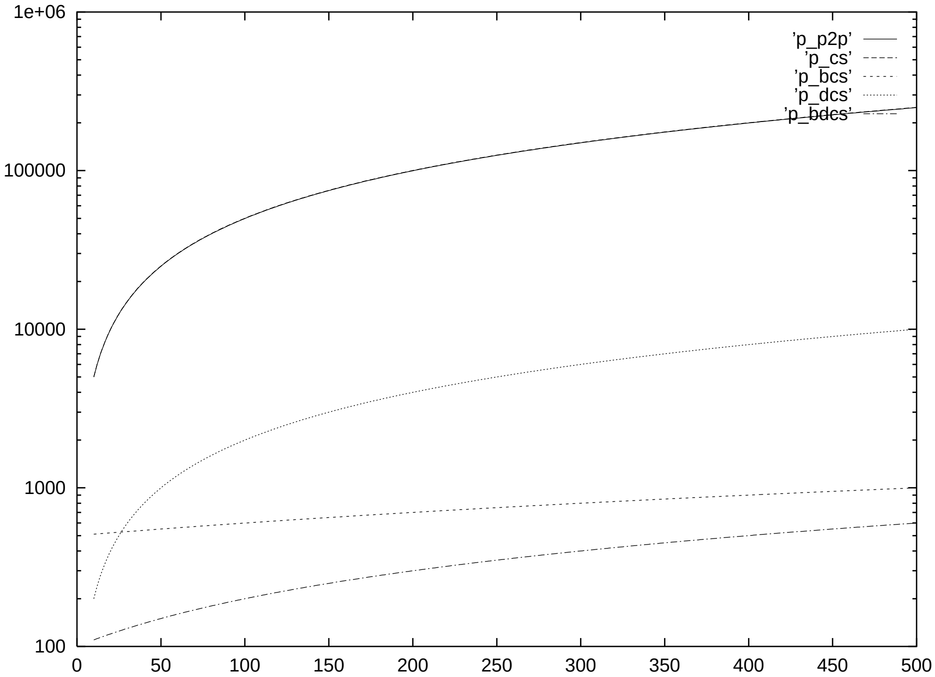
4.2 ForceLets

ForceLets were developed to improve the positional coherency of virtual objects following a path through space-time while using the minimum amount of network traffic. To test the capabilities of the ForceLet algorithm, an experiment was conducted with three candidates:

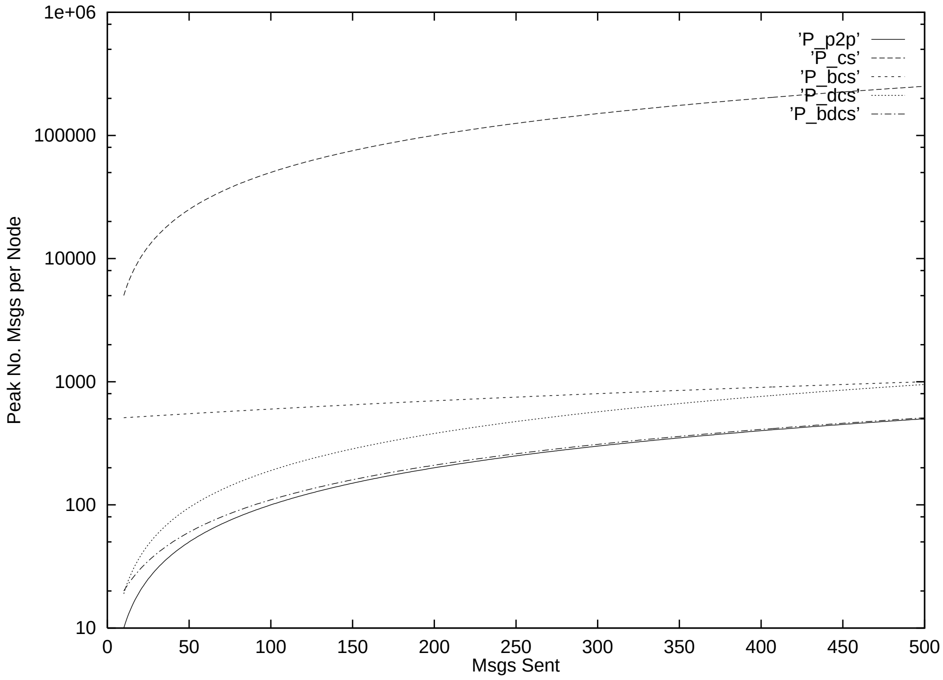
1. The *stream of data* algorithm that simply samples the source position at regular intervals (10 Hz), and transmits it to the destination.
2. The *dead reckoning* algorithm that uses velocity information to predict future positions in time. Only when the predicted position differs from the true position by a threshold does an update occur.
3. *ForceLets* as described in Section 2.3 using a special ForceLet type for sinusoidal paths (as one would expect for orbits).

The algorithms were required to follow a source path through one-dimensional space. Figure 7 shows the source curve, and resulting paths for the algorithms. For this simulation, the network delay was set at 20 ms, the stream of data sampling frequency was 10 Hz, and the dead reckoning threshold was 0.5.

The *stream of data* algorithm shows quantization effects due to the sampling frequency and a time offset due to the network delay. This causes a compounded lag and jitter effect. The *dead reckoning* algorithm fairs better, however still shows a jitter effect when the threshold is exceeded. The *ForceLet algorithm*



(a)



(b)

Fig. 6. Evaluation of network: (a) shows the total number of packets in the network, and (b) shows the largest number of packets for a node in the network

shows a small delay at the start of the path due to network lag, after which the curve follows the source curve exactly.

Numerical analysis of the results are summarized in Table 1. Notice that the ForceLet algorithm has the least average error, least maximum error, and lowest number of updates.

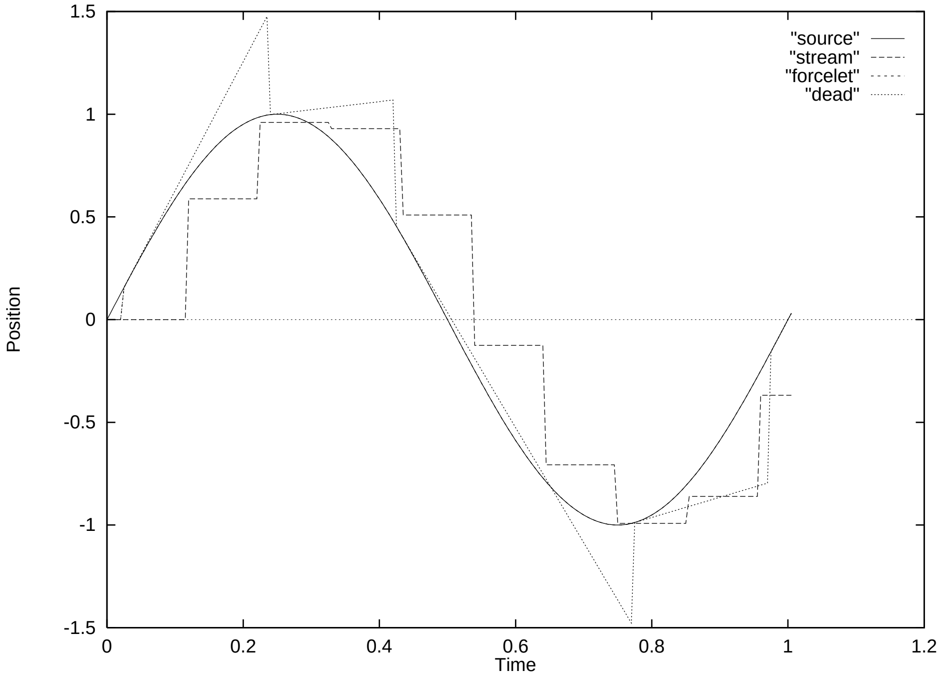


Fig. 7. Graph of coherency algorithms. The network delay was set at 20 ms, the sample frequency for stream of data was set at 10 Hz, and the threshold for the dead reckoning algorithm is set at 0.5

An additional experiment was conducted to determine the capacity of the simulator for three test cases[12]: (1) a varying number of ForceLets attributed to a single object, (2) a single ForceLet attributed to a varying number of objects, and (3) a varying number of objects with eight ForceLets attributed to each. Up to 4096 ForceLets were simulated and timed on an idle SPARCStation 20.

The results of the experiment show that the computational complexity of virtual object simulation scales linearly with the number of ForceLets at a rate of 615 thousand ForceLets/Second up to the 4096 ForceLets. Using a 10 Hz refresh rate, this translates to a capacity for 61.5 thousand ForceLets on an idle node.

Table 1. Comparison of Coherency Algorithms

Algorithm	No. of Updates	Max. Error	Avg. Error
Stream	15	0.727	0.258
Dead Reckoning	6	0.609	0.146
ForceLet	1	0.125	0.00155

4.3 Rendering Engine

Tests have been performed on the NAVL prototype to evaluate the collision detection mechanism, and determine the number of active virtual objects the prototype can handle before the rendering engine overloads the system.

The test examines the average overhead (in ms), percentage of total CPU time, and simulation refresh rate for a varying number of non-colliding virtual objects. This test was performed on an idle SparcStation 20. The simulation refresh rate is held at a constant 20 Hz until the overhead exceeds available CPU resources, at which time it drops.

Table 2. Rendering engine results. The average overhead is the amount of time spent generating the state buffer and detecting collisions per frame. The CPU percentage notes the amount of CPU required on a SPARCStation 20. Note that the framerate is held constant at 20 FPS until the CPU usage exceeds the systems capacity

Number of Virtual Objects	Average Overhead (ms)	CPU (%)	Average Framerate (FPS)
16	1	0	20
32	2	0	20
64	7	0	20
128	28	40	20
256	105	100	9.4

Notice that the overhead increases dramatically between 128 and 256 virtual objects. This is expected given the poor performance of nested-loop collision detection algorithm. While reasonable for small virtual worlds, clearly the rendering algorithm must be improved to support larger environments.

5 Summary and Future Work

This paper shows the design and implementation for a distributed virtual reality system, built from the ground up to improve network efficiency, object distribution and coherency, system resource management, and overall performance.

A prototype of the NAVL concepts has been recently completed, and the evaluation of its capabilities are underway. Preliminary results show that the fundamental systems provide a good network architecture and simulation environment, however the collision detection mechanism limits the system to small virtual worlds (less than 256 virtual objects).

References

[1] C. Carlsson and O. Hagsand. DIVE – a multi user virtual reality system. In *IEEE VRAIS*, pages 394–400. IEEE, 1993.

[2] Circle of Fire Studios, Inc. http://www.activeworlds.com.

[3] G. Couris and J. Dollimore. *Distributed Systems: Concepts and Design*. Addison–Wesley, Reading, Massachusetts, 1988.

[4] IEEE standard for distributed interactive simulation — application protocols. IEEE std 1278.1–1995, IEEE, 1995.

[5] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics: Principles and Practice*. Addison–Wesley, Reading, Massachusetts, 2nd edition, 1990.

[6] Philip M. Hubbard. *Collision Detection for Interactive Graphics Applications*. PhD thesis, Department of Computer Science, Brown University, October 1993.

[7] Rick Kazman. Making WAVES: On the design of architectures for low-end distributed environments. In *IEEE VRAIS*, pages 443–449. IEEE, 1993.

[8] Rick Kazman. HIDRA: An architecture for highly dynamic physically based multi–agent simulations. *IJCS*, pages 149–164, May 1995.

[9] Chris Shaw and Mark Green. The MR toolkit peers package and experiment. In *IEEE VRAIS*, pages 463–469. IEEE, 1993.

[10] Abraham Silberschatz and James L. Peterson. *Operating System Concepts*. Addison–Wesley, Reading, Massachusetts, 1988.

[11] The virtual reality modeling language 2.0. ISO/IEC cd14772, VRML Architecture Group.

[12] Martine Wedlake. NAVL phase 1 results: Simulation engine. Technical report, Department of Electrical and Computer Engineering, University of Victoria, February 1998.

[13] Martine Wedlake and Kin F. Li. Sailing the high seas with the NAVL virtual reality system. In *IEEE WESCANEX*, pages 208–213, Winnipeg, M.N., May 1997. IEEE.

[14] Martine Wedlake and Kin F. Li. The simulaiton and execution architecture for the NAVL DVR system. In *IEEE PACRIM*, pages 93–96, Victoria, B.C., August 1997. IEEE.