Distributed I/O for Virtual Environments

Andreas Engberg and Brent Lehman,{andreas, brent}@create.ucsb.edu
HPDM Lab, CREATE, Department of Music
University of California, Santa Barbara, USA

I. Introduction

Virtual Environment technoligy (VE), also known as Virtual Reality (VR), has been around for quite a while, since the first flight-simulators in the early 1960s up to today's complete simulations of war and 3D games. The major difference between the earlier systems and the VE-applications of today is the fact that nowadays simulation takes place in real-time and allows the user to participate in the virtual world without any specialized systems or hardware. More and more VE applications focus on supporting multiple users in worlds where users interact with each other over a network, regardless of whether the application is a game, a chat room, or a traffic simulation. The traditional techniques for interactions have been the monitor, the keyboard, and the mouse, but more advanced tools and hardware exist for interaction between users. Our research at the UCSB HPDM Lab concentrates on tools such as head mounted displays (HMDs), motion trackers, gloves and wands. The I/O devices are linked to local hosts, which means that the host has full control of the sensor. As more interactions are needed, more sensors are used and need to be distributed to other applications in the network. In an ordinary system where the sensor is tightly coupled to one application, that program stores the values internally and broadcasts the sensor values to all other nodes in the system. This approach has a major disadvantage: the sensor is owned by one and only one application and thus we risk the behavior of the entire system if a sensor halts or becomes unavailable.

In this report, we will show techniques to make these VE sensor devices available over a network, so that different programs can use the sensors simultaneously without making significant changes to the applications and without risking the entire system if a sensor crashes.

II. Background

As part of the ATON project [10], a high-performance multi-user distributed VE is needed. The design calls for mixing real world information from cameras [13] with information obtained from various user I/O sensors such as gloves, motion trackers, etc. Users in the virtual world will be able to control real robots [14] to perform different tasks. The approach means that our software has to satisfy the following criteria:

- Independent of hardware platforms and operating systems.
- Different programming languages.
- Easy to use.
- Real-time capabilities.

Previous work on the project has included finding appropriate I/O devices [3],modeling tools [4], creating large 3D models [7] and testing different rendering tools [5].

A. Test environment

Our test environment consists of two off-the-shelf Intel PCs (one high-end and one mid-scale), two SGI Octanes, and three SUN servers (two with Intel x86 PC coprocessor cards built in). This gives a total of nine computers with different specialties. The operating systems include Windows NT and several version of UNIX (Solaris, IRIX, Linux), and a 100Mbit network connects all machines. The available input devices are two six degrees-of-freedom (DOF) magnetic trackers [2], one 2DOF glove with gesture capturing software [8], and standard mouse and keyboard. Furthermore, a stereo-optic head mounted display (HMD) [9] allows us to do more enhanced and realistic testing of the system. By using the HMD, the effect of network latency and jitter is immediately obvious to the user.

III. Distributed sensors

The goal of a distributed sensor system is to let a sensor be available to more than one application (client) simultaneously, see fig 1. The distribution can either be done by letting the sensor send out data to all clients that are interested in the data, or by letting each client query or poll the sensor whenever appropriate. Both approaches have advantages and disadvantages. The first approach

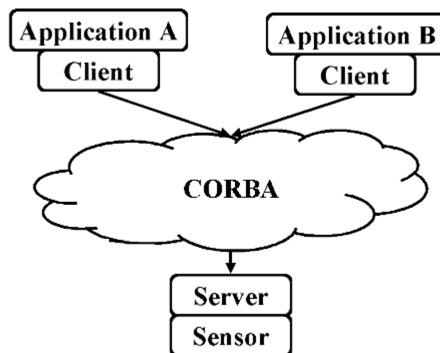

Fig. 1. By introducing distributed servers, different applications can read the same sensor as if they had exclusive access to it. The communication between the clients and servers is embedded into the system and is hidden from the end-user.

has the advantage that the sensor (server) is a stand-alone

program, and only notifies the clients when a new data has been produced, usually by using a publish/subscribe mechanism. This can cause problems, though, if the sensor updates its data frequently and thus uses more network bandwidth than necessary. On the other hand, the latter technique allows a client to query or poll data from a server only when necessary, but this approach is more vulnerable to collisions caused by several clients trying to access a sensor server at the same time.

In our research, we have focused on the approach where the server updates the clients based on the occurance of a specific event; all clients are still operational regardless of whether the sensor for some reason halts.

One major disadvantage with a distributed sensor is the latency and the jitter introduced by adding a layer between the sensor and its clients. A naive way to solve this would be to increase the bandwidth and throughput of the medium. This can be extremely costly and even impossible, thus more sophisticated techniques are needed. Another minor issue is that overhead is added to clients even when they are running on the same node as the server. A solution to this would be to create customized nodes that read a sensor directly instead of over a network, but that means unnecessarily distributing knowledge of sensor details (see section VI-A).

IV. Multiple hardware platforms and programming languages

Since our ATON VE application requires multi-platform support in a distributed environment, the decision was made to use the Common Object Request Broker Architecture (CORBA), which is a standardized protocol for distributed objects. Like any software endeavor, a virtual world is easier to build and manage as a module than as a tightly integrated part of a larger system. CORBA readily facilitates such a structure by supporting a client-server architecture and encapsulating the inter-object interface code on both sides. The CORBA implementation used most heavily in ATON is MICO [11] , which is a free and open-source version of CORBA.

One of the keystones of CORBA is the Naming service. When a server is created, it registers itself in the nameserver. A client will then be able to query the nameserver where to find a object with a certain name.

To let servers and clients communicate independently of the operating system and the hardware platform, a description of their interface is created in the so-called CORBA Interface Definition Language (IDL) (see fig 2). The IDL description file is compiled on the desired platform, and the result is two templates of code representing the server and the client, respectively.

Even though this technique has the advantage of using an independent specification language, it also has one major drawback; whenever the interface between the client and server is changed (the IDL file), all files related to the previous templates must be recompiled using the new templates. The penalty for revising the interface grows with the changes of the interface, hence a well-designed interface is very important to avoid future complications.

```
// CORBA IDL description for the Sensor interfaces

// A sensor must be of one of these types.

enum SensorType {
   SENSOR_1DOF,
   SENSOR_2DOF,
   SENSOR_3DOF,
   SENSOR_4DOF,
   SENSOR_5DOF,
   SENSOR_6DOF,
   SENSOR_GLOVE,
   SENSOR_GLOVE_1DOF,
   SENSOR_GLOVE_2DOF,
   SENSOR_GLOVE_3DOF,
   SENSOR_GLOVE_4DOF,
   SENSOR_GLOVE_5DOF,
   SENSOR_GLOVE_6DOF
};

// A description of a sensor, including name, type and version.

struct SensorInfo {
   char          info[20];
   char          version[5];
   SensorType    type;
   char          nrdata;
};

// Only the raw sensor values are sent over the network.

struct SensorData{
   double        data[11];
};

// The actual interface of a server.

interface Sensor {
   short         Start();
   short         Stop();
   short         GetInfo(inout SensorInfo info);
   short         GetData(inout SensorData data);
};
```

Fig. 2. The interface between clients and servers are described in the Interface Definition Language. In this case, one enumeration (which lists the sensor types), two data structures (for data and information sending), and the different functions that a server supports.

The interface between the sensor and a client (shown in fig 2) consists of just four functions; Start, Stop, GetInfo and GetData. The client can use Start and Stop to inform the server to begin or stop updating a client. This can best be described as a subscribe/unsubscribe functionality, since a server will be able to keep track of how many clients are currently using the produced data, and can stop publishing data if no listeners are using the service.

A client can get information about what kind of sensor a server actually represents. Even though this is not necessary for a client to know, it can still be helpful. The information tells a client which type a sensor is (e.g. DOF, glove data available), along with the name of the server, version information, and how much data the server will produce. The type of sensor (enumerated in SensorType) does not exactly describe a sensor, but rather gives a "hint" of what kind of information to expect for the sensor. For

example, a sensor with three distinct positions will be identical to a sensor with three orientations since both sensors have three DOF.

The sensor data is retrieved using the GetData function. This function is either implemented as a stream (invoked by the server, located in the client), or a polling function (invoked by the client, located in the server). Regardless of which approach is used, the interface will still look the same.

NOTE: The keyword *inout* in the IDL example is not supported by all IDL compilers and is therefore a bit risky. Our suggestion is to use *out* parameters, or to use two variables (*in* and *out*) instead.

V. Communication between sensors and applications

Whenever an application wants to use a sensor, it must create an instance of a client and tell it to talk to the correct server. This process can be tricky unless the programmer has CORBA experience and knowledge of the different parameters that need to be set. In order to make the client more user-friendly, a generalized "wrapper" was created (see fig 3). Basically, this wrapper hides most of the CORBA-related configuration and server handling, and thus allows the programmer to use a client with minimal configuration.

The functionality of the client is more or less the same as the server: it has the ability to start and stop the server, and to get data and information from the server. However, the information from the server (GetInfo in the client-server interface, see 2) has been divided into three parts: Version, Info, and QuerySensor. This was done to remove the somewhat confusing data structure SensorInfo and dissolve its parts into three sub-functions.

An application that uses a sensor client will always get its data in the data structure DistSensorData, regardless what kind of sensor is attached in the other end. This makes it very easy to switch between different sensors without changing the application code, since the position part of the data will always be at the pos-position in the structure, and so on. Also, the transition from one sensor to another can be made at run-time without any problems (assuming that the new sensor is available and operational).

In order to minimize the latency in a distributed sensor even more, the communication between the client and application can be divided (see fig 4). The application will always be in non-blocking mode, even though the client is getting new data from the server. This may cause other problems, described in section VII.

VI. Current status

Even though our approach can model any 6-DOF sensor (with 5 extra parameters, e.g., for gloves), it would be useless unless we have a concrete scenario in which to test the different sensors.

In this section, a description of our system will be given along with brief overview of the sensors we are using (currently eight). Also, a few test scenarios will be explained.

```
// Generic C++ definition for sensor clients
// A generic 3D vector is always useful
typedef struct _Vector3D_{
   double x;
   double y;
   double z;
} Vector3D;
// Data read from a glove
typedef struct _VectorGlove{
   union{
      struct{
         double thumb,index,middle,ring,pinky;
      };
      double fingers[5];
   };
} VectorGlove;
// A generic datatype containing data about a position,
// an orientation and a glove. All values can be read
// directly from the raw-portion of the datatype.
typedef struct _VSensorData{
   unsigned int type;
   union{
      struct{
         Vector3D      pos;
         Vector3D      ori;
         VectorGlove   glove;
         };
      double   raw[11];
   };
} DistSensorData;
// DistSensor is the client of a sensor. It is similar
// to the interface defined in the IDL but is more
// userfriendly.
class DistSensor{
public:
   DistSensor(char *servername,int argc, char *argv[]);
   short              Start();
   short              Stop();
   char*              Version();
   char*              Info();
   bool               Read(DistSensorData &data);
   SensorType         QuerySensor();
protected:
   Sensor_var         client;
private:
   struct SensorInfo info;
};
```

Fig. 3. A special client was created (in C++) to hide most of the CORBA related initializations and server communications.

A. System

The system currently in use supports nine servers and four applications (see fig 5). Although the system has not been tested with maximum load (all applications running at the same time), it is worth mentioning that the system works well, though there is a noticeable decrease in performance when one server is frequently used. Nine different sensor servers are implemented, where three are physical sensors (one glove and two birds) and six are different software simulations (five splineservers and one Paramics simulation server). The Flock of Birds (FOB) tracker server and the Paramics interface are running on two SUN workstations (with the Solaris OS), while the spline servers and the glove server are running on two PCs (with WindowsNT).

The glove provides position tracking with 2 DOF (limited pitch and roll), as well as the inflection of the fingers. Pitch and roll can only operate $\pm 60^o$ and the inflection is

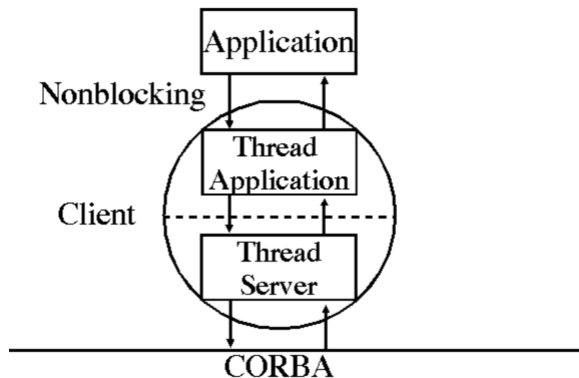

Fig. 4. The internal of the client and server is divided into two threads in order to increase performance.

recalibrated by the sensor server to the range of 0.0 (fully stretched) and 1.0 (fully bent). The glove is very difficult to use for orientation since whenever the rotation around one axis is more than 60° it starts to affect the other axis as well. This makes the glove useful only in the cases where the fingers are of interest.

On the other hand, the 6 DOF magnetic trackers are operational up to two meters from the transceiver, and give position and orientation relative to this transceiver (the origin). The position has been calibrated to meters and orientation in $\pm 180^o$. An application can later, if necessary, scale the position internally. One of the 6-DOF sensors is connected to an HMD and thus monitors the movements of the head (position and orientation).

Distributed sensor servers do not only hide the sensors from the clients, they also allow us to simulate various types of sensors in software. For instance, a key issue in the ATON project is to simulate the flow of traffic by using a simulation software package called Paramics. The traffic flow follows certain routes, and we can simulate incidents, congestion, and other traffic-related problems. While waiting for Paramics to arrive and be configured, a "fake" Paramics sensor was created. This sensor reads spline curves from a file and can be used to generate traffic routes similar to Paramics. In fact, we created five different routes, and each one was represented by a separate sensor server. This implementation gave valuable information about how many cars (e.g., the number of sensors) our simulation could handle, and also showed how easy it was to create new customized sensors.

B. Software

Four separate programs are used to test and demonstrate the functionality of different sensors. The applications can be run simultaneously, each one with different scenarios, on different hardware platforms and operating systems, and they are completely independent of one another.

Two of the programs, EON [6] and VRUT [12], are off-the-shelf VE rendering products, whereby EON is an advanced commercial product, and VRUT is an easy-to-use "freeware" 3D renderer. The software packages use almost identical test worlds, and the result is very similar in both programs. One interesting test would be to run the two programs at the same time, with identical worlds and let one application render the right eye and the other application render the left eye.
The testworld contained twenty different cars, each one connected to a spline server, and a "world" model consisting of an area of several hundred yards square around the east entrance to the UCSB campus. The viewport was controlled by the two FOB sensors and the dataglove.

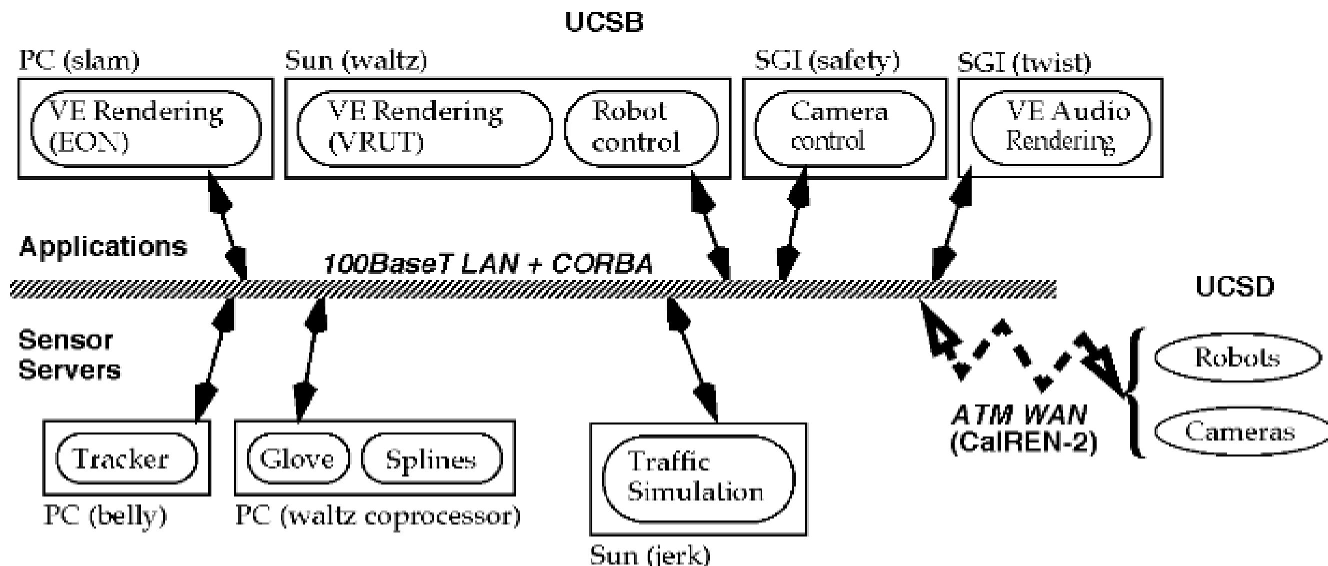

Fig. 5. The system that is currently used. It contains nine servers and four applications, and runs distributed on five different machines.

B.1 EON

Although both VRUT and EON were used as VE delivery systems, the requirements of ATON made us focus more of our research on the EON system, basically because it is more advanced and allows more customizations.

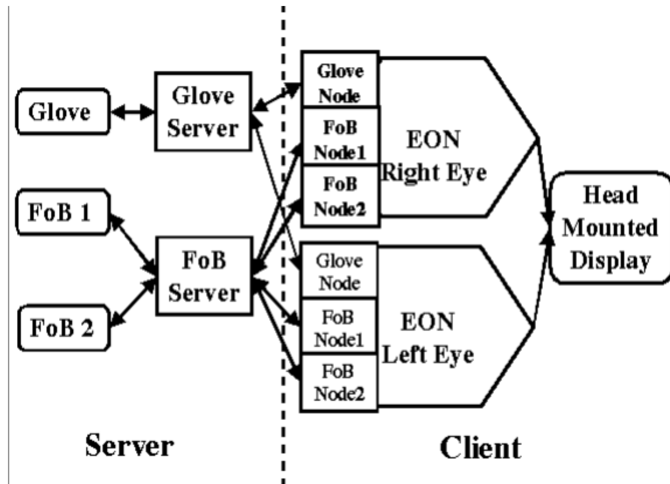

Fig. 6. The layout of EON. The different nodes are DLLs that communicates with the servers over a network.

The clients, in this case, are Dynamic Linked Libraries (DLLs) running atop EON. In the nomenclature used by EON, these DLLs are known as "nodes." The nodes we developed serve as liaisons between the servers and EON by controlling the motion of a camera through the virtual environment. An overview of the system is shown in fig 6. As in the general case, one FOB tracker is attached to the HMD, but the other sensor was attached to the glove.

A special navigation system was developed that allows users to control their motion through space by pointing their hands in the direction they want to go and making a fist. The tighter one flexes one's fingers, the faster one goes. The user may also look in any direction without affecting his/her flight path. You accomplish this simply by turning your head.

In order to enhance the realism of the virtual world, a stereo-optic renderer can be used. This is achieved by presenting two slightly separated images to the two eyes. To achieve stereoscopy, the EON simulator runns on two separate workstations. Each runs the same environment and talks to the same server programs, but renders scenery from a slightly different virtual viewpoint from the other simulator. The user can adjust the location of each of his/her virtual eyes through a keyboard interface.

Unfortunately, the framerate was constant but below real-time (30 frames per second). The usage of all servers and interaction with a user still felt realistic, however. One of our models of the campus area uses a huge texture image, which, when rendered with an older-vintage videocard, decreases the framerate below real usability. A low-resolution texture image improved the framerate, but also decreased the quality of the world.

B.2 Real-world interaction

Two real-world applications were created, one for viewing and controlling remote-controlled pan/tilt/zoom cameras and one for controlling remote robots. (Note that these devices were installed at our partner campus in San Diego, several hundred miles from us.) The software for interacting with the mechanical hardware (robot and camera) is web based, using Java servlets or HTTP protocols for the wide-area interchange. On top of this software, a CORBA layer was added that interacted with our sensors (using the sensor server architecture, of course).

The robots [14] are controlled by the Saphira [1] program. It uses the orientation of the available FOB sensor (not the one mounted on the HMD) to navigate the robot in the real world. The sensor's pitch and roll are used to determine the velocity of the robot's motion.

In order to display real-world information from cameras, a fourth program is used. The software is a simple image renderer where the image is a picture taken by a camera mounted on the robot and sent over the network as a JPEG file. A sensor, in this case a 6 DOF tracker mounted on the viewer's HMD, controls the perspective of the camera (pan, tilt and zoom). This allows a user to wear the HMD and control the camera with head movements.

VII. Conclusion

Several different techniques exist to create distributed object applications, from low level highly customized network wrappers, to more generic and costly solutions. A server can support a client with data either by letting the client query the server for data, or by letting the server stream the data to the client. Which approach to choose depends on the situation; if alot of data is produced by the server and the connection to the client is slow, a polled solution is appropriate. If, on the other hand, the client needs to be invoked as soon as the server has produced a new value, then a streaming approach can work better. A streaming sensor is very suitable in a virtual environment due to two factors:
1) a client needs to receive data with as little delay as possible to minimize overall system latency, and 2) avoid conflicts and delays if two or more sensors try to access a sensor server at the same time.

Since, in our case, the final demo included only a few distributed sensors (less than ten), and only a handful of applications, the latency was in fact worse with the streaming approach than the polling version. In order to improve the performance, the polling version was therefore used. This shows that the application alone does not determine which approach to use, but that the number of sensors and clients can also have a significant impact on the system.

Latency jitter is not very noticable in this system, but when it occurs, it immediately delays all applications using distributed sensors. Even though it is almost impossible to predict jitter and latency, there are different approaches to minimize their impact on the system. A history buffer can be used to simulate the behavior of the sensor when the current data is not up to date, but this approach can only

be used for a short period of time and with caution, and when correct data eventually comes, the drift needs to be as low as possible.

VIII. Future work

Even though the ATON project is coming to an end, several interesting topics have been found that we consider worth investigating in the future. As a next step, we hope to move the sensors and the servers to a real-time operating system and thus improve the predictability and scalability of the servers. We also plan to integrate more clients, such as spatial sound processors and new hardware control interfaces to enable new applications of this architecture.

References

[1] ActiveMedia Inc. *Saphira Software Manual*, 2000.
[2] Ascension Technology Corporation. *The Flock of Birds, Installation and Operation Guide*, January 1999.
[3] Howard Durand. User input and output devices for advanced software systems. Technical report, CREATE. Department of Music. University of California. Santa Barbara, 2000.
[4] Andreas Engberg and Howard Durand. A comparision of 3d modeling programs. Technical report, CREATE. Department of Music. University of California. Santa Barbara, December 2000.
[5] Andreas Engberg, Howard Durand, Gilroy Menezes, and Brent Lehman. A comparision of virtual environments. Technical report, CREATE. Department of Music. University of California. Santa Barbara, December 2000.
[6] EON Reality Inc. *EON Studio*, 2000. www.eonreality.com.
[7] Stephen Travis Pope et al. Create aton world-building tasks. Technical report, CREATE. Department of Music. University of California. Santa Barbara, November 2000. www.create.ucsb.edu/aton/0010/UCSD.Model.html.
[8] Fifth Dimension Technologies. *5DT Data Glove 5, User's Manual*, February 2000.
[9] Kaiser Electro-Optics, Inc. *ProView 30, Owner's Manual*, 1997.
[10] HPDM Lab. Aton project. Technical report, CREATE. Department of Music. University of California. Santa Barbara, 2000. www.create.ucsb.edu/aton.
[11] Arno Puder and Kay Römer. *MICO. An Open Source COBRA Implementation*, volume 3. Morgan Kaufmann Publishers, 2000.
[12] Research Center for Virtual Environments and Behavior. University of California. Santa Barbara. *VRUT Manual*, August 1999. www.recveb.ucsb.edu/www/index.html.
[13] Mohan Trivedi, Brett Hall, Greg Kogut, and Steve Roche. Web-based teleautonomy and telepresence. *45th SPIE Optical Science and Technology Conference, Applications and Science of Neural Networks, Fuzzy Systems, and Evolutionary Computation III, San Diego*, 2000.
[14] Jim Wilson, Stefano Coleden, Mike Fillon, and Scott Gourley. Road robots to the rescue. *Popular Mechanics*, 2000.