



The ATON Project

Center for Research in Electronic Art Technology
University of California, Santa Barbara
Computer Vision and Robotics Research Laboratory
University of California, San Diego
CalTrans Test-Bed Center For Interoperability



ATON Report 2001.06.5: The Distributed Processing Environment for High-Performance Distributed Multimedia Applications

Stephen Travis Pope, Frode Holm, Ahmi Wolf, and Francisco Iovino
{stp, frode, ahmi, francisco}@create.ucsb.edu
CREATE, UCSB, June, 2001

Contents

- Preface2
- Introduction2
- HPDM Applications: An Example2
- The DPE architecture3
- RIDL: The Real-time Multimedia Interface Description Language5
 - HPDM RIDL Example5
 - Actual RIDL Parameters6
 - RIDL Compiler and Compiler-Compiler7
- The NodeManager Service7
- The DPE Service Interface9
- DPE Databases10
- The DPE Manager Tool10
- Summary11

Preface

Our group is involved in implementing large-scale multimedia software for application areas ranging from multi-user virtual worlds to complex real-time sound synthesis. We call this class of system *High-Performance Distributed Multimedia* (HPDM) software. The *Distributed Processing Environment* (DPE) is an infrastructure for configuring and managing HPDM software. It consists of several components that allow the start-up, monitoring, and shut-down of software services on a network. This report describes the design and implementation of the prototype DPE system, which we built for the ATON project.

Introduction

In our previous work, we have built several generations of distributed multimedia virtual environment systems. We have used various techniques for managing distributed real-time programs, including low-level socket programming with proprietary protocols, and the CORBA standard. In our current project, we have considered the higher-level infrastructure necessary to reliably deploy complex multi-server real-time applications over LAN- or even a WAN-based systems using CORBA. We call the infrastructure and tools we built the *Distributed Processing Environment* (DPE); it helps us describe, install, start-up, monitor, and shut-down large-scale HPDM applications such as our multi-user virtual environment framework.

In the sections below, we will introduce the basic architecture of DPE-managed HPDM applications, and then detail the Real-time Multimedia Interface Description Language (RIDL) and its compiler, and the DPE management infrastructure and end-user tools. We should note at the start that the DPE implementation is in a “proof of concept” stage; we have built the main infrastructure components and databases, but several pieces (mainly the *DPE_Service* implementation) are yet to be completed. This report focusses on the motivations, architecture, and design of the system.

HPDM Applications: An Example

In our previous HPDM design report (ATON Report 2000.06.1), we presented several example scenarios of HPDM applications. Since then, we have built and demonstrated a multi-site virtual environment that uses the HPDM architecture. The basic configuration of this application is shown in the Figure below.

The items on the left are the user interface for the virtual environment, including a head-mounted display with head- and hand-tracking and (optional) spatialised sound output to the user. The right-hand side shows the servers that provide the inputs to the “world,” meaning both the static and mobile cameras and the simulation programs that can move objects within the world. The boxes in the center are the visual and aural renderers. These components communicate with one another via CORBA messages, and all are managed and monitored (also via CORBA) by the DPE manager shown at the bottom of the Figure.

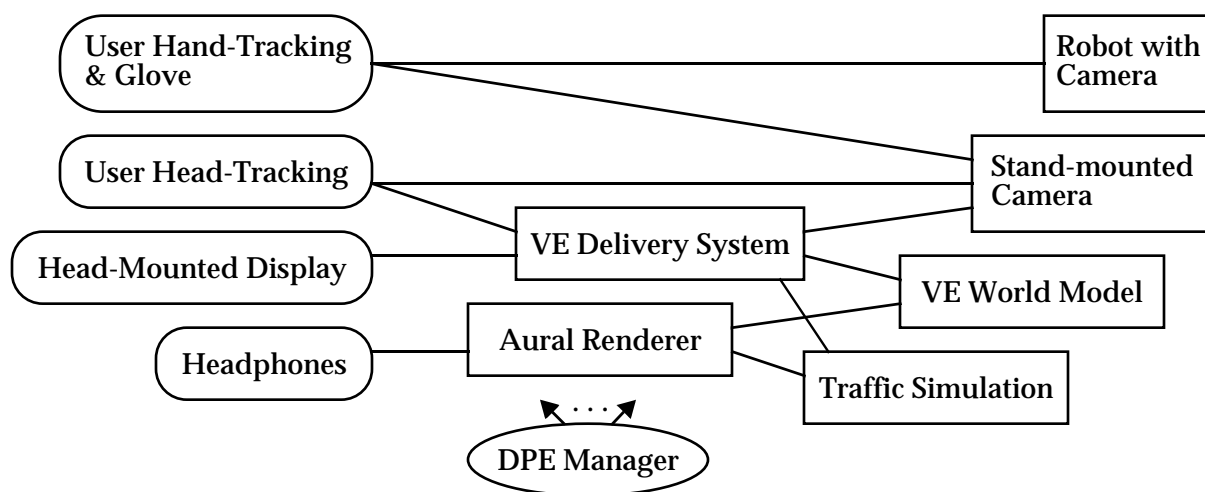


Figure: Example HPDM Application Configuration

The DPE architecture

The basic DPE infrastructure consists of four components:

- 1) a node manager service that runs on every machine in the network;
- 2) a set of methods to be implemented by DPE-managed CORBA services;
- 3) a set of databases describing the network components and the features of the software we wish to execute; and
- 4) a DPE manager program and associated GUI.

These components communicate with one another using the CORBA software standard for distributed processing. We introduce each of the components in the following paragraphs, and present the details in the later sections.

Every machine in the network runs a small DPE *NodeManager* service; this process controls and monitors relevant resources on the node level and presents them to the DPE manager. The *NodeManager* program supports remote messages for (a) sign-of-life tests and regular heartbeats, (b) basic resource and system activity queries, and (c) spawning new services. (Details will be given below.)

Each DPE-managed service implements (in addition to its native service interface) a CORBA interface called *DPE_Service* that includes the sign-of-life and heartbeat calls, and also provides for controlled shut-down and restart of the service. These programs use the CORBA-based Real-time multimedia Interface Definition Language (RIDL) to describe their objects and methods, as well as the quality-of-service and run-time infrastructure requirements of the services. This allows the *NodeManager* to control services running on the node, and also lets the DPE Manager migrate services between nodes at run-time and thus balance the load of the system.

The DPE databases store information describing the available computing resources and the “scenarios” that detail the configuration of a specific HPDM application. Scenarios may

include information about services that require certain hardware or need to run on specific machines in the network.

The DPE manager acts as the brain of the system; it presents a GUI that can be used to configure, start, monitor, and stop the HPDM application, and can use the run-time information derived from the RIDL descriptions to automatically distribute and tune HPDM applications. The user can also manually control and fine tune applications during runtime.

The Figure below illustrates the DPE architecture. At the center is the DPE Manager program, which communicates with its databases and GUI (on the left), and with all DPE-managed nodes on the network (on the right). On each hardware node in the system, there is assumed to be a running CORBA ORB, and a NodeManager program, which provides a CORBA service for node management. If there are any DPE-managed CORBA services on the node, they will implement a CORBA service interface so that they can be monitored and stopped/restarted by the DPE Manager. Other CORBA-enabled programs (clients of the DPE-managed service) also communicate through the node's ORB to send messages to objects within the service program; only the DPE manager generally talks to the NodeManager or the service's DPE_Service interface.

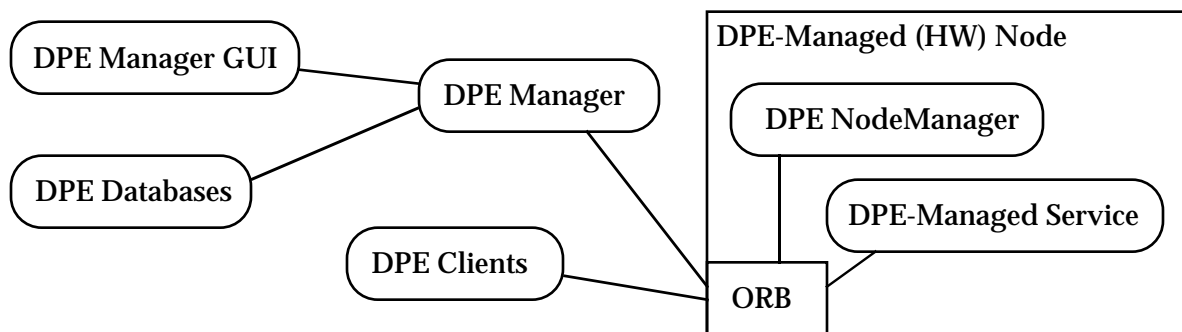


Figure: The DPE Architecture

To run an HPDM application using DPE, an ORB and a node manager service must be present on each machine. The DPE manager is started by hand and reads a system configuration from the database. It sends calls to the appropriate node managers to start their heartbeats (sending regular activity messages to the DPE manager). The DPE manager then proceeds to send start-up messages to the node managers to spawn their various services as described in the scenario. It tells the services where to find the centralized CORBA name server, and what names they should register themselves under in the naming service (so that their clients can locate them).

During execution of the system, the DPE manager monitors the nodes and services to make certain that they are still active, and that the load on the machines (e.g., in terms of CPU cycles or network I/O) is within operational limits. Thus, the DPE can be used to address the issues of fault tolerance and load-balancing, though these concerns are secondary to basic system start-up and shut-down in the initial implementation.

RIDL: The Real-time Multimedia Interface Description Language

To create a DPE-enabled application, we need to extend the CORBA IDL description of the service interfaces with Quality-of-Service (QoS) annotations. To provide for this, we have extended the CORBA IDL with a series of QoS-related parameters in what we call the Real-time multimedia IDL, or RIDL. We have built a first-generation RIDL compiler that uses a relational database as its interface repository, and have developed several HPDM scenarios for testing the RIDL/DPE system.

The RIDL extensions address several areas that the CORBA interface definition language (and even the recent real-time and messaging extensions) does not address. RIDL allows us to incorporate quality-of-service (QoS) annotations and run-time information into CORBA interface descriptions. The multi-pass RIDL compiler reads standard IDL and generates source code files in the language of choice (C++, Java, or Smalltalk), and takes the extended method annotations and stores them in a database for later use by the DPE run-time system that manages the applications.

To support large HPDM applications, there are five kinds of extensions that we identified to be integrated with CORBA IDL definitions:

- 1) application-level and network-level quality-of-service (QoS) information (allowable latency and jitter, dropped packet ratio, out-of-order packet handling policy, etc.);
- 2) method performance characteristics (e.g., run-time scaling with data size);
- 3) interface hardware requirements (assumptions about special HW);
- 4) method run-time characteristics (e.g., expected call rate); and
- 5) server implementation characteristics (e.g., what platforms can it run on).

The RIDL example below shows one method from the IDL interface that would be used by a spatial sound processor; it includes the standard CORBA IDL for the method's function signature (return type, name, and parameters), as well as the QoS annotation that will be processed by the RIDL compiler. (Note that we have not included in this example the exceptions that might be raised from this call.)

HPDM RIDL Example

```
// Sound source placement method: take a sound source (streaming object),
// the source and user position/orientation, and the geometrical room model, and
// fill-in the spatial sound placement object with psychoacoustical processing information.

void place_source(           // function name and return value
                        // in parameters (can be passed by value):
    in sound_generator a,    // the streaming sound object
    in point_6D source_position, // the source's position and orientation
    in point_6D listener_position, // the user's position and orientation
    in room_geometry room,   // the model of the room's reflecting surfaces

                        // inout parameter (passed by reference or copied out):
    inout sound_source placement, // the spatialized sound object
```

```

// (We may want to describe the exceptions raised by the call.)

// RIDL function description and QoS
called = EVENT_LIKE,           // called irregularly
frequency = USER_IO,         // called on user (or source) movement
                                // proc. time scales as the square of room complexity
processing_demands = ON2(room),
max_buffer = 40,              // can be 40 msec or so ahead of real-time
max_jitter = 10,             // needs less than 10 msec latency jitter
late_policy = DROP_UPDATE,    // drop the call if it's too late
);

```

The RIDL annotations look rather different from the standard CORBA IDL; they are key-value pairs (as in `max_jitter = 10`) because we have to set the values of the method's QoS properties. This means that there is no straight-forward way to have their syntax look "normal" to a standard IDL compiler. The two options are either to have the RIDL compiler strip them out before handing the CORBA IDL off to the second-pass compiler, or to embed the RIDL annotations in legal CORBA IDL comments (e.g., preface them with a special token such as `/*_`). In the prototype RIDL compiler, we chose the second of these options.

Actual RIDL Parameters

The following is a list of the QoS parameters that we have designed to date, grouped into the five categories we introduced above. The RIDL compiler and support facilities make it easy to extend this list in the future.

QoS parameters

```

latency (usec) -- allowable or expected
latency_typical (usec)
latency_max (usec) -- allowable
jitter (%) -- allowable
dropping { allowed, not_allowed }
ordering { required, optional }

```

Performance characteristics

```

in_size_typical (bytes)
in_size_max (bytes)
out_size_typical (bytes)
out_size_max (bytes)
processing_dependency { input, output, sum, product }
processing_scaling { const, orderN, orderN2, orderExp, orderNLogN }

```

Hardware requirements

```

sound_i/o { dac, adc, multi-chan }
graphics_i/o { screen, camera, hmd }
haptic_i/o { glove, tracker, mouse }
network_controller { 10BaseT, 100BaseT, 1000BaseT, OC3, OC12 }

```

Run-time characteristics

bandwidth_typical (kBytes/sec)
bandwidth_max (kBytes/sec)
call_format { regular, irregular, scheduled, background }
call_frequency { per_sample, per_frame, conditional, user_mvmt, user_io }

Server characteristics

source language { C, C++, Java, Smalltalk }
platform processor { x86, SPARC, MIPS, PPC }
platform OS { Solaris, Linux, IRIX, MS-Windows, Macintosh }
preferred ORB { TAO, Orbix, MICO, SmalltalkBroker/DST }

RIDL Compiler and Compiler-Compiler

The RIDL extended IDL compiler reads RIDL annotation and creates records in a relational database for each method in a RIDL-extended IDL interface. This database is then used by the run-time DPE system. We have defined SQL tables for this; they act as a “RIDL annotation repository.”

The RIDL “compiler-compiler” is a simple Smalltalk program that generates the RIDL parser (in C) and the database table formats (in SQL) based on a description of the kinds of annotation we need to support (described in a C header file). Thus, a single file describing a formalized list of the QoS parameters listed above is used to generate the RIDL compiler itself, and the databases that will store the derived RIDL annotations. The RIDL compiler currently runs as pre-processor for the MICO and TAO IDL compilers.

The NodeManager Service

As introduced above, there is a small service program that must be running (in addition to the ORB) on each node that is to be managed by DPE. The DPE manager talks to the NodeManager to find out if the node is running, and how busy it is. To start a new service, the DPE manager simply sends a shell command string and CORBA naming server identifier to the appropriate NodeManager. A simplified version of the CORBA IDL for this service is shown below.

```
// NodeManager IDL for the HPDM DPE
// This service will be running on all DPE-managed nodes.
//
interface NodeManager {

// Basic ping (sign of life) and heartbeat management
    boolean ping();
    void setHeartBeat(in short interval, in string dpeReference);
    void setNameServer(in string NameServerReference);

// Performance and Resource queries
    short availableMemory();
    short availableCPU();
    short availableNetworkIO();
```

```

// Starting services
void startService(in string command, in string name);
};

```

The most basic query from the DPE Manager to the NodeManager is the sign-of-life query ping(), i.e., “are you alive?” The DPE Manager can also request that NodeManagers send in regular heartbeat messages (on the order of every few seconds). These two facilities give us the opportunity to build some measure of fault tolerance (or at least pro-active fault detection) into the system. The resource and performance queries allow the DPE Manager to ask nodes how busy they are; this way, the DPE Manager can perform dynamic resource-based service-to-node allocation and simple load-balancing. The final call in the interface is used by the DPE Manager to tell a node to start running a new DPE-managed service. These are typically started using a command-line string and a name for use by the naming service. The DPE Manager uses this call to spawn services on the nodes of the network when bringing up an HPDM application.

The NodeManager is the “kernel” of DPE. Below is pseudo-code for its operation. We have implemented the bulk of this for the Linux platform.

```

// DPE NodeManager pseudo-code
//
// The purpose of ReadyService is to make sure the containing process of the service is running
// and to generate the object reference, but *not* necessarily to create it the object (the servant).
// The DPE Manager will thus be able call this method for each NodeManager without having to
// worry about dependencies between services, i.e. all such dependencies will be handled during
// the 2nd stage of initializing, when all object references are known.

```

```

NodeManager::ReadyService() {
    try {
        <start process if not already running>
        <wait for 1st stage init to complete. (maybe use time-out semaphore?)>
        <signal should also include indication of success>
        if <not ok> return failure;
        <get object reference from database/naming service>
        return ok;
    } catch (...) {
        return failure
    }
}

```

```

// 2nd stage init. This could just as easily be called directly from the DPE Manager, since it now
// has the object reference.

```

```

NodeManager::InitService(service_var service) {
    ok = service.InitService(); // service object will be created here and if successful will
    // enter the READY state

    return ok;
}

```

```
// The run method is called when 1st and 2nd stage init has been called successfully and
// the service object is in the READY state.
// short-cut: DPE or NodeManager could have called each one separately.
```

```
NodeManager::RunAllServices() {
    for <all services readied> {
        service.Run();
    }
}
```

```
// This is an outline for the main of the process hosting a service
```

```
main(...) {
    ....
    <create object references (but not the servants if they depend on other services)>
    <publish in database/Naming Service>
    ....
    <try to obtain hardware resources if applicable>
    ....
    <signal result of 1st stage init to NodeManager>

    orb->Run();
}
```

The example above illustrates the logic of the multi-stage NodeManager service startup sequence. The main() function is typical of CORBA applications; it does some minor set-up and then calls the ORB's start routine to begin CORBA message processing.

The DPE Service Interface

Each CORBA service program that is to be managed by the DPE also has to implement a small interface that allows the DPE Manager to poll its state and to stop/restart it. The IDL for this interface is given below. Note that servers (as well as nodes) can have heartbeats that they are told to send to the DPE manager at regular intervals, and that services can (theoretically) be politely shut down and restarted.

```
// Generic Service IDL for the HPDM DPE
// This will be implemented by all DPE-managed CORBA services.
//
interface Service {

    // Basic ping (sign of life)
    bool ping();
    void setHeartBeat(short interval, string dpeReference);

    // Starting/stopping services
    void shutdownService(short delay);
    void restartService(short delay);
    void killService(short delay);
};
```

To date we have only built “custom-hacked” versions of DPE service programs, but most of the implementation will be straightforward (based on the NodeManager implementation. Because they manage system facilities and processes, both the NodeManager and the DPE_Service interface are quite operating-system-dependent.

DPE Databases

To capture and use this information, we need to maintain several kinds of databases or repositories that are derived from RIDL descriptions and auxiliary information. These include:

- 1) IDL interfaces (data structures, function signatures);
- 2) RIDL annotation (IDL extensions, QoS, etc.);
- 3) network configuration (machines, MIPS, network, HW, I/O, etc.); and
- 4) scenarios (distributed application configurations).

We are currently using a simple relational database for this, and both the RIDL compiler and the DPE manager share access to it. (The same database tables used by the DPE manager also serve as our object naming service in the current implementation.)

The IDL interfaces are stored as output files; we do not currently use a CORBA interface repository. The tables for the RIDL interface repository are automatically generated by the RIDL compiler compiler. The network configuration and application scenario database schema were generated manually.

The DPE Manager Tool

As the final process in setting up the DPE framework, we have built a “proof-of-concept” (but functional) version of the DPE management tool. A screen dump of this initial version is shown in the Figure below; one can see three main panes in the view:

- top: the table of available server nodes with their characteristics (the data in the table comes from the network configuration database and the node’s NodeManager service);
- middle: the list of running services on the selected machine (DPE manages this); and
- bottom: the text of recent messages from the selected service (retrieved from the running service itself).

There are pop-up menus (not fully implemented) in the three table views that support operations on the node, service, and message lists. Future versions of this tool will allow users to start and stop individual services and entire HPDM scenarios from this tool. The other tool that remains to be written is a “scenario configuration editor.” To change the network configuration database, we currently have to edit the SQL files, though we have debated extending the NodeManager’s interface to include run-time queries for this kind of information.

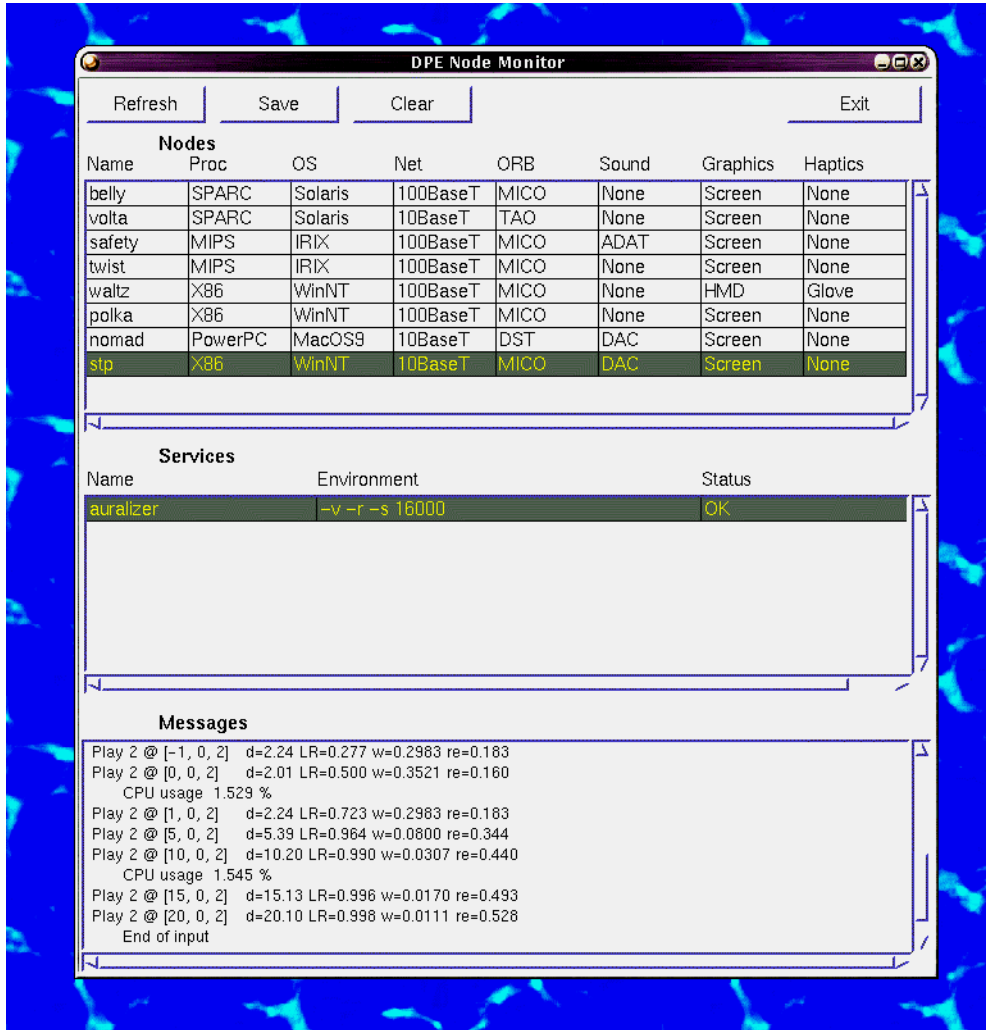


Figure: DPE Manager GUI

Summary

The goal of this effort is to enable us to build flexible and scalable real-time applications that run across several computers on a local- or wide-area network. In the ATON Project, we have deployed a complex system that uses between three and nine geographically distributed computers, and incorporates mobile robots, streaming control information and video, and real-time simulation programs.

The DPE framework provides us with facilities to describe the requirements and behavior of HPDM systems, and to manage them at run-time; it will make it much easier to extend this to the next generation of complexity, and to build and tune future implementations.