Application Development and Management in The Programmers' Playground

T. Paul McCartney, E.F. Berkley Shands,
Kenneth J. Goldman, William M. Shapiro

WUCS-98-18

June 1998

Department of Computer Science
Washington University
Campus Box 1045
One Brookings Drive
Saint Louis, MO 63130-4899

Application Development and Management in The Programmers' Playground

T. Paul McCartney, E.F. Berkley Shands,
Kenneth J. Goldman, William M. Shapiro
Department of Computer Science
Washington University
St. Louis, Missouri 63130, USA
{paul | berkley | kjg | wms1}@cs.wustl.edu
http://www.cs.wustl.edu/cs/playground/

Abstract

Application management refers to the process of making software applications available to end-users and providing automated mechanisms for launching and joining such applications. The Programmers' Playground is a computing environment for creating distributed applications from modular, reusable components. This paper discusses a set of tools that enable application developers to: (1) design and debug Playground distributed applications from existing "off-the-shelf" components using a visual configuration tool, (2) make new application components available on the Internet through a "launcher" service, and (3) make complete distributed applications available via a World Wide Web interface, enabling end-users to launch and join the applications interactively. Our system automates the tasks involved in launching distributed applications. This system is completely general-purpose; no programming is required to customize it for particular applications.

Keywords: distributed computing, distributed debugging, resource allocation, visual configuration, World Wide Web

1. Introduction

Distributed applications supported by a global electronic infrastructure such as the Internet have tremendous potential for providing users with customized communication and computation environments. Applications include remote collaboration, information and resource sharing, and access to broadcast media. A problem with making these applications available to end-users is the technical issues involved with launching, coordinating, and configuring the application's components. First, for general Internet applications, components should be capable of running on many different types of workstations. For high performance components, written in a programming language such as C++, there needs to be operating system specific code responsible for performing operations such as load monitoring and process creation. This code may be either within the components or within a separate launch system. Difficulties often arise when this code must be ported to different computing platforms. Second, for multi-user applications there needs to be some type of a shared coordination mechanism that is accessible by the application's users. The World Wide Web (Web) is a natural choice for providing global accessibility to such information. However, it should not be necessary for each application developer to implement complex Web mechanisms for their applications. Third, the communication structure of a distributed application must be established upon launching the application. For applications that consist of purely modular, reusable components, this communication structure is likely to be externally configured rather than hard-coded into the components. General-purpose, automated configuration systems facilitate the development of such applications. Finally, for optimal performance, some type of intelligent resource allocation should be performed in order to make the best use of system resources with respect to the performance requirements of application components. Again, this functionality is best implemented as a separate, general purpose system rather than being hard-coded into the application code.

This paper presents a set of tools for creating and using distributed applications. On the creation side, we present a visual configuration tool for dynamically establishing interprocess communication among application components. This tool empowers application designers, who may or may not be skilled in textual programming, to create distributed applications from existing general purpose components without the need for programming. This tool also provides distributed debugging functionality, enabling one to visualize and control an unmodified application's communication (i.e., it is not necessary to recompile application components in order to perform debugging tasks). Once designed with this tool, an application specification may be saved. We also present a distributed application framework for making completed applications available to end-users given the specification. This framework includes a Web-based graphical front-end for specifying application launch and join operations and a system for performing resource allocation, launching, and configuration of distributed applications. This system is completely general-purpose and reusable; no programming is required to customize it for a particular application.

The remainder of this paper is organized as follows. Section 2 presents related work that others have done in the areas of distributed debugging and distributed application resource allocation. Section 3 presents The Programmers' Playground, a programming environment for creating distributed multimedia applications. Playground was used to implement the functionality that is presented in this paper. Section 4 discusses a couple of motivating example distributed applications. Section 5 presents the system's user tools for visual configuration and debugging of distributed applications as well as the Web interface for using completed applications. Section 6 presents the Brokerage System architecture, the heart of the system that automates the resource allocation, launching, and configuration of Playground distributed applications. Finally, Section 7 ends with a brief summary and discussion.

2. Related Work

In the area of distributed debugging, Kunz [7] implemented a time-line oriented visualization system for use with the Hermes distributed computing language [18]. Kunz identified two complementary ways of visualizing a distributed computation: the *process view* and the *temporal view*. In the process view, a distributed application is represented as a graph where processors are shown as vertices and communication pathways are shown as edges. Our visual configuration tool (Section 5.1) uses this approach, and augments the process structure diagram with real-time animation and interactive control mechanisms. The temporal view, which Kunz used in his implementation, uses a time-line to show how the various processes of an application communicate over time. This approach is excellent for "postmortem" analysis of an application's execution. To simplify the temporal view, Kunz also implemented the concept of "Process Clusters" for condensing the execution visualization of large distributed applications down to a more comprehendible display. However, this work did not provide a way to interactively control the execution of live distributed applications for the purposes of debugging.

The IVD system [6] was developed to provide an integrated approach to debugging, performance analysis, and data visualization for message passing parallel applications. Conventional, single-process debuggers and visualization tools can be used without modification with IVD to debug parallel applications. IVD uses ESP, Event Sense Protocols, to intercept windows-based commands from a centralized debugging interface to various process debuggers. Our system also supports interactive debugging of multi-process applications. Like IVD, application components may be debugged in our system without modification or recompilation. However, we provide only a high level, communication oriented debugging tool rather than support for the conventional source code oriented debugging that IVD provides.

Paralex [1], [2] is a programming environment in which parallel programs can be developed and executed on distributed systems as if they were running on uniform, parallel multiprocessors. Like our work, Paralex programs are specified using a graphical language that is based on coarse-grain components. Unlike our work, communication in Paralex is based on a strict dataflow model, where the computation of a component is enabled only when each of its input tokens have been supplied. Paralex uses the dataflow approach as a means to implement interactive debugging of a distributed application. When debugging, a special component called the "controller" is added to the application with links to each other application component. The controller sends "enabling tokens" to each component based on user interaction, giving the user the ability to suspend execution, insert breakpoints, or single step through the program execution. Our debugging mechanism also provides suspend and single step functionality (breakpoint functionality is planned, but has not been implemented). However, while Paralex requires the recompilation of application components in order to utilize debugging functionality, our system allows developers to debug application components without modification.

OpenUI [13] is a user interface management system designed specifically for client/server and three-tiered enterprise distributed applications. OpenUI applications consist of a graphical front-end, application logic, and a distributed database management system. The OpenUI development environment includes a graphics toolkit, a messaging architecture, a visual builder, and OpenWeb, a mechanism for launching user interfaces for OpenUI. To make a client/server application available to users on the Web, developers create a link on their Web page that points to the downloadable user interface specification. Web users who have installed the OpenUI "helper application" can then download user interface specification, using the helper application as the graphical front-end of the (already running) application. The application management functionality described in this paper is much more general, allowing distributed applications consisting of an arbitrary number of components to be launched and dynamically configured. The DCE Web [8] applies a set of distributed computing technologies, based on OSF DCE, to solving the authorization and organizational problems of the Web. The DCE Web also incorporates a name-service based mechanism for locating objects.

A number of techniques have been tried to deal with distributed systems and large resource bases. PVM [19] and PRM [12] both address the distributed access problem by employing a hierarchy of control agents to allocate jobs to processors. The application management mechanism described within this paper is similar to PRM. Like PRM, we have chosen a hierarchical model of resource control. PRM has a hierarchy of different types "managers" that are used to launch the components of a distributed application; these managers are similar to the brokerage system described in Section 6. The main differences between our approach and theirs is in the user interface, application specification, and configuration. PRM does not provide a mechanism that allows end-users to design and launch applications; we provide a visual configuration tool and an intuitive World Wide Web mechanism. In this way, our applications can be easily developed and distributed to the entire Internet community. In PRM, the "job manager" is used to decide how to allocate processes to machines. The way of specializing how the components of an application are allocated is to customize the job manager (i.e., through programming). Our method of specifying the application is generic (i.e., no programming), allowing the application designer to state preferences about each module's desired performance that the system will attempt to enforce. Finally, PRM has no built-in configuration mechanism; it only allocates jobs to machines. Our system performs the configuration among the modules of a Playground application, facilitating the use of general purpose reusable components. In addition, we also support a specialized configuration mechanism for client/server applications, connecting client modules to server modules. No programming is required to specify this communication structure.

3. The Programmers' Playground

The Programmers' Playground [3], [4], [5] is a software library and run-time system that supports a simple model of distributed computing called *I/O abstraction*. I/O abstraction provides a separation of computation from communication that is well-suited for end-user construction of customized distributed applications from computational building blocks. Playground users do not need to write any source code to establish communication between the components of a distributed application, nor do they need to understand the details of how communication occurs. This section provides an overview of Playground; details on Playground may be found elsewhere [3], [4], [5].

In Playground, a distributed application consists of a number of communicating components called *modules* which are implemented as separately running processes. Each *module* has a data boundary[1] containing *published variables* that may be externally observed and/or manipulated by other application modules. Modules are created by writing C++ programs using the Playground software library. This library provides a set of publishable data types, including base types (e.g., integer, real, string), tuples, and aggregates (e.g., arrays, lists). Programmers may arbitrarily nest these types to form new publishable data types, and new publishable aggregates may be defined as well.

A distributed application consists of a collection of independent modules and a communication configuration of *logical connections* among the published variables in the module data boundaries. Whenever a module updates one of its own published data items, the new value is implicitly communicated to all connected variables in other modules. The details of how the communication is handled are hidden from the implementor and users of the module. This simplifies module construction and gives the run-time system flexibility in optimizing communication. The configuration of connections is determined dynamically at run-time, rather than statically at compile time. This gives end-users the flexibility to create new applications on the fly by plugging together existing off-the-shelf components.

I/O abstraction communication is declarative, rather than imperative. That is, one declares high-level logical connections among the data items of individual modules, as opposed to directing communication within the control flow of the module. Output is implicit, a by-product of computation. Input is observed passively, or handled by reactive control within a module. This declarative approach simplifies application programming by cleanly separating computation from communication. Software modules written using I/O abstraction do not make explicit requests to establish or effect communication, but instead are concerned only with the details of the local computation. Exposing the configuration also allows the run-time system to handle communication more effectively.

Playground has been implemented on a number of different computing platforms including Sun Solaris, SGI Irix, DEC OSF, Windows NT, Linux, and NetBSD. Playground's interprocess communication implementation enables modules to communicate seamlessly among the various platforms. For example, one could execute an application that consists of communicating modules running on Solaris, OSF, and Windows NT. While the Java programming language also provides seamless multi-platform

1. In other papers describing the I/O abstraction concept, the data interface of an I/O abstraction module has been called the "presentation." Since this dissertation deals with user interfaces, we use the term "data boundary" in order to avoid confusion.

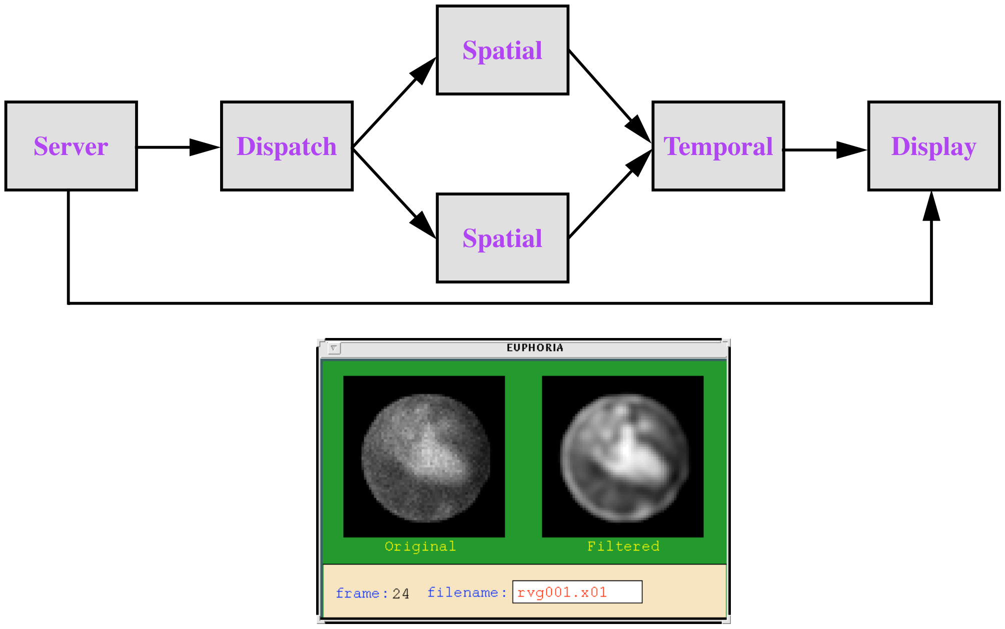
communication, this capability comes at the price of efficiency. Playground offers the benefit of running fast, natively compiled component modules. While RPC-based systems such as CORBA [21], [22] also provide multi-platform communication for natively compiled components, the communication structure of RPC components tends to be tightly coupled in comparison to Playground components. That is, RPC components have explicit references to external components embedded within their source code, making it difficult to separate out and reuse components in different contexts. In contrast, Playground modules operate exclusively in terms of a local state with no explicit references to other components. It is our hope that Playground's modular nature, efficiency, and multi-platform capabilities will enable developers to exploit Internet-wide available computing resources in the creation of high performance distributed applications.

4. Motivating Examples

This section presents two example applications that motivate the need for application management functionality. Section 4.1 discusses a distributed medical image filtering pipeline which requires intelligent resource allocation to achieve optimal performance. Section 4.2 talks about a shared video conferencing application which demonstrates the need for an automated client/server architecture.

4.1 Medical Image Filtering

A nuclear medicine radioactive blood pool study is used to create movies of a beating human heart for diagnostic purposes. Each movie consists of a series of images taken by injecting the patient with a radioactive compound and observing the heart with a "radionuclide ventriculogram" apparatus. However, the images suffer from noise caused by ambient radiation, making them difficult to read. This section describes an application for digitally filtering these images using an image processing pipeline.



frame: 24 filename: rvg001.x01

Figure 1: Medical image processing pipeline.

In this example, the digital filtering operation consists of two stages. First, a *spatial filtering* operation is applied. In this operation, each pixel of an image is processed in terms of its surrounding pixels. Second, a *temporal filtering* operation is applied to the result of the spatially filtered images. In this operation, each pixel of an image is processed in terms of the corresponding pixels of the previous and following images of the series.

An interactive filtering application is created through the use of a number of communicating modules (Figure 1). Spatial and Temporal filter modules are created independently, each having input and output image published variables. Since the spatial filtering operation is computationally intensive, a Dispatch module is used to distribute the images between two identical Spatial modules running concurrently on separate processors. The spatially filtered images are ordered at the Temporal module, filtered, and sent to the user interface display module. This user interface was constructed interactively with EUPHORIA, an end-user oriented user interface management system. The display is used to specify the name of the image series and to view both the original and filtered image series.

Since this application is designed as a processing pipeline, obviously the modules should run concurrently on separate processors. Running all of the modules on the same processor would severely hinder performance since none of the modules would run in parallel. Each module has different performance characteristics that should be considered in deciding how to allocate modules to available processors. The Spatial module is the most computationally intensive, therefore its two instantaneous should be run on the fastest two processors with the lightest loads. The Server and Dispatch modules just forward image information among modules, and therefore do not need to execute on fast processors. However, the network connections among these modules should be fast (e.g., running the Server on a workstation in Japan and the Dispatch on a workstation in Brazil would result in slow performance). The user interface display module is usually fastest running on the user's local workstation due to fewer communication delays among the X Windows client and server portions [16]. All of these factors should be considered when launching this application. This paper includes a description of a "brokerage system" (Section 6) that is used to perform resource allocation and automated launching of this and similar applications.

4.2 Vaudeville

Vaudeville is an ATM-based teleconferencing application that supports scalable, multi-party video teleconferences [14]. Each conference participant uses special hardware called The MultiMedia eXplorer (MMX) [15], along with an associated video camera and microphone, to send, receive, and view NTSC-quality live video. In addition, each participant has a graphical user interface (Figure 2) that displays still images of all conference participants, the currently selected speaker, and various controls (e.g., volume, mute).

Vaudeville is an example of a client/server application (Figure 3). The server portion consists of modules that maintain information about the conference participants and the currently selected speaker. The client portion consists of modules that are specific to an individual participant. When a user wants to join a conference, it is necessary to launch the client modules and configure the client to the appropriate server modules. Each client module has different performance requirements. For example, a client has an MMX (MultiMedia eXplorer) module that must run on the user's local machine in order to access the MMX hardware.

Clearly, it is undesirable to expose the end-user to the details of resource allocation, launching, and configuration of the client modules. Ideally, we would like to make the application available to end-users through a publicly accessible medium and automate its start-up. This paper describes a set of tools that

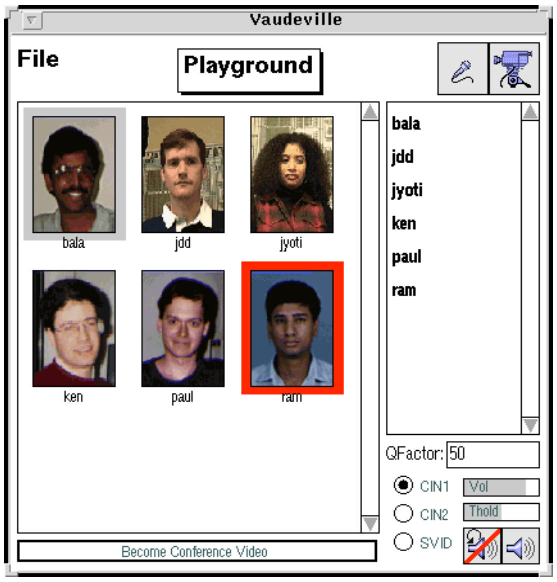


Figure 2: Vaudeville Video Teleconferencing GUI.

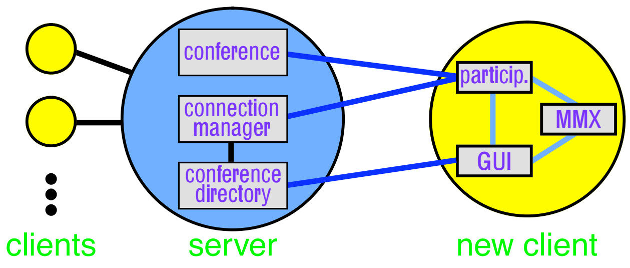


Figure 3: Modules of a Vaudeville conference server and client.

allow designers to create this application from existing components, make the application available to end-users through a web-based interface, and automate its resource allocation, launching, and configuration.

5. User Tools

A major goal of our system is to empower users to construct and use Playground distributed applications, without requiring textual programming. This section describes the Application Management System's user tools: the Mediator, a visual configuration tool for application design and debugging, and the Liaison, an applet for using an application through a Web-based interface.

5.1 Mediator

The Mediator tool provides a graphical user interface for designing Playground distributed applications. With this tool, application designers can:

1. interactively launch or kill modules
2. view the published interfaces of modules
3. dynamically configure communication among running modules
4. perform communication-oriented debugging tasks
5. save completed application specifications

Figure 4 shows the Mediator visual interface with the configured Medical Image Filtering application (Section 4.1). The Mediator is implemented as a special type of Playground module known as a *configurer*. This allows the Mediator to examine and visualize the published variables of other running modules (as well as changes to these interfaces and the modules' running status). Modules are displayed as boxes within the Mediator's "Configuration Area," where each module is represented as a title bar with a number of published variables attached below. For example, in Figure 4, the TEMPORAL module is shown as a box with its "IN" and "OUT" associated published variables.

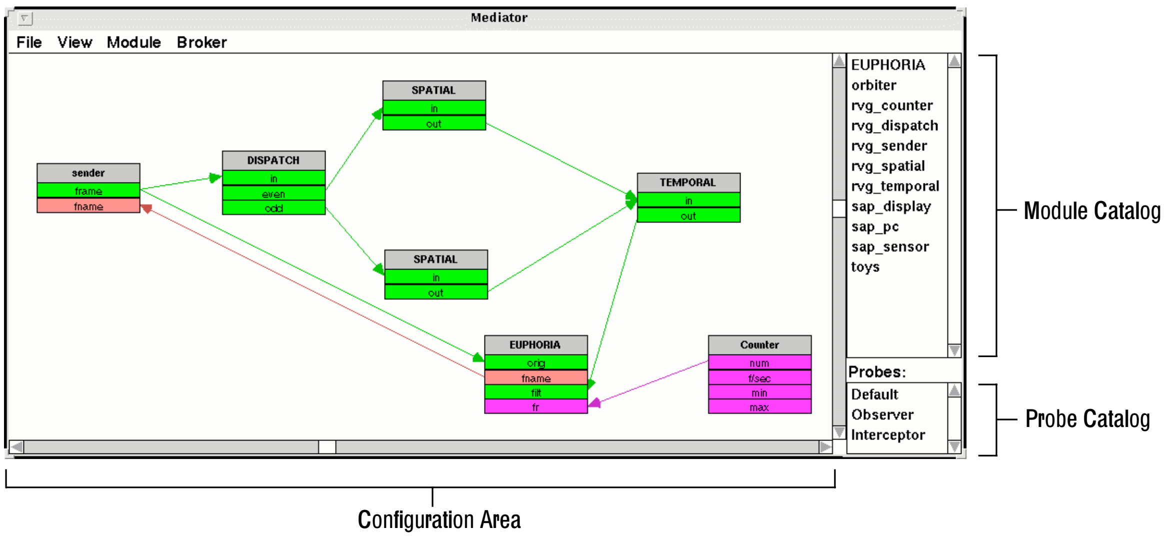
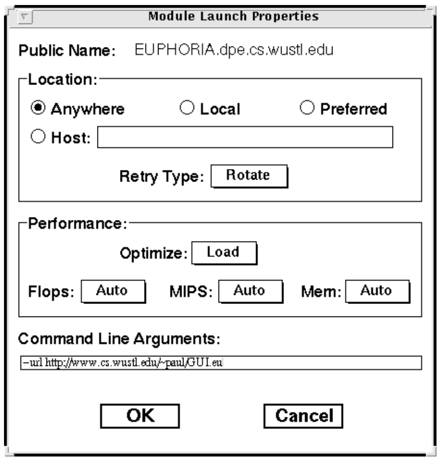


Figure 4: Mediator configuration window, showing the Medical Image Filtering application.

Application designers can interactively create applications from existing modular components. A "Module Catalog" pane in the Mediator window provides a list of launchable modules. This list is obtained by the Mediator through communication with a separate "Brokerage System" (Section 6). Designers launch modules by simply dragging the module name from the Module Catalog into the Configuration Area. Prior to launch, the designer may optionally specify module launch preferences (Figure 5) such as command line arguments, and how to optimize module placement (e.g., select the workstation with the lightest load). Based on this information (or default information) the Brokerage System locates the requested module, determines which workstation to use, and launches the module on that workstation, returning reference information back to the Mediator. Given this reference, the Mediator is able to visualize the module, perform configuration tasks, and terminate the module.

Given a number of running modules, the application designer can interactively configure communication among by simply drawing connection lines between the modules' published variables. The Mediator uses information about the modules to aid the designer in performing the configuration task. For example, the Mediator highlights variables that are type/permission compatible while the designer creates a new


Module Launch Properties

Public Name: EUPHORIA.dpe.cs.wustl.edu

Location: [✓ Anywhere] Local Preferred Host:

Retry Type: Rotate

Performance:

Optimize: Load

Flops: Auto MIPS: Auto Mem: Auto

Command Line Arguments:

–url http://www.cs.wustl.edu/~paul/GUI.eu

Figure 5: Module launch preferences window.

connection. Once the designer has completed the application, a specification of the modules and connections may be saved for use either locally or through the Liaison web interface (Section 5.2).

Distributed Debugging

In general, conventional debuggers are not well suited for debugging distributed applications since they tend to be oriented toward the source code of a single process whereas a distributed application consists of multiple processes that communicate. Although it is possible to apply a conventional debugger on each process of a distributed application, this approach can be rather cumbersome and difficult to manage. Also, debugging a distributed application in this way does not give one good feel for how information is flowing among application components -- knowledge that is crucial in debugging distributed applications.

The Mediator includes a *probe* mechanism for performing communication oriented debugging tasks. From the Probe Catalog (Figure 4), the end-user can drag probes on the various logical connections of a Playground distributed application. Probes are shown as an LED-like displays centered on the connection (Figure 6). When a value update is communicated along a connection, the probe blinks to reflect the transmission. This blinking gives the designer a high level view of the application's communication pattern.

The Mediator supports two types of probes: *observer* and *interceptor*. When an observer probe is associated with a connection, the original connection is not disturbed. Instead, the observer probe passively receives updates from the "upstream" published variable of the connection[2] whenever it is modified, independent of the monitored connection. Observer probes are only meant to give a feel for how the application communicates, without actually affecting its timing characteristics. Since the connection's update and the probes observation are independent, the observer probe may blink sometime before or after the "downstream" variable has actually received the update. In contrast, when an interceptor probe is placed on a connection, the connection's communication is actually rerouted through the probe (i.e., the probe "intercepts" the updates). While this can slow down the communication along the connection, this approach offers a couple of advantages. First, the blinking of the probe represents the probes receipt of the update; hence this indication always precedes the actual delivery of the update to the downstream variable

2. Probes may receive updates from both the upstream and downstream variables in the case of bidirectional connections.

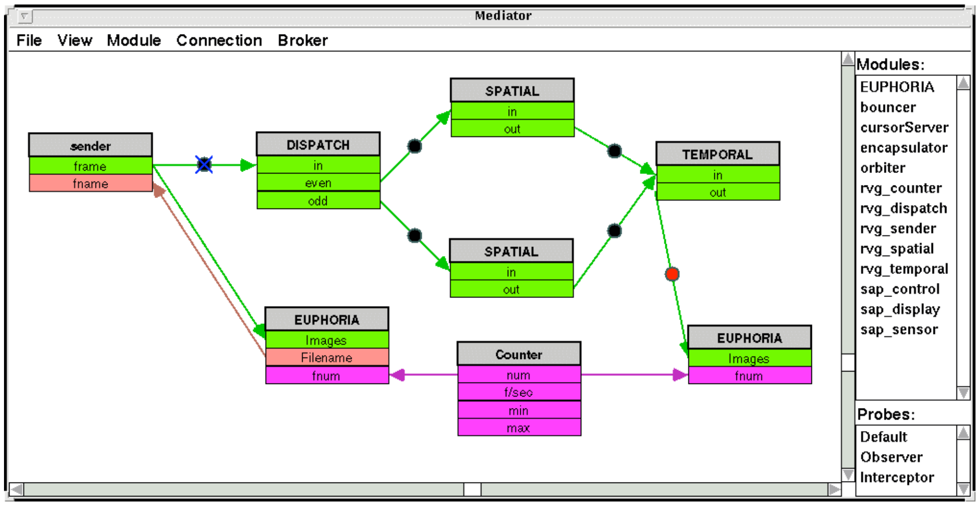


Figure 6: Medical Image Filtering application, with debugging probes.

of the connection. Second, intercepting the communicated values gives one the opportunity to control the delivery of the connection's updates.

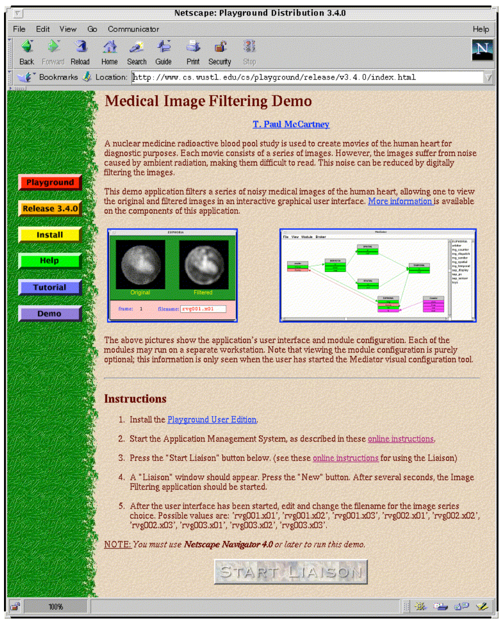
With interceptor probes, communication may be controlled in a few different ways. One can specify a delay associated with the delivery of value updates, having the effect of slowing down application execution. This helps the developer to observe and make sense of the communication patterns of the application. Also, delayed update delivery provides a means of testing the application under different running conditions, simulating situations such as slow network/workstation performance. This type of testing can be helpful in finding timing "race conditions" that often plague distributed applications. A connection's communication may also be suspended, enabling the developer to essentially halt the application and interactively single-step through the update delivery.

For example, Figure 6 shows probes applied to various connections of the Medical Image Filtering application (Section 4.1). A probe is placed on the connection between the "Sender" and "Dispatch" modules. The "Sender" module sends a series of images into the pipeline for processing; this connection is suspended, shown as an "X" over the connection. As described earlier, half of the images are sent to one instance of a "Spatial" module and the other half goes to another instance. The results of this filtering is sent to the "Temporal" module which gathers images and eventually filters each image when its surrounding images have been obtained. Probes allow the application developer to see how images are communicated through the application. In this case, since the first probe is suspended, the developer can single step through the delivery of each image to the Dispatch module. By observing the probe animation, the designer can see that every other delivered image takes "the high road" (i.e., Spatial filter #1) or "the low road" (i.e., Spatial filter #2) through the pipeline. Also, one can see that several images need to be received at the Temporal filter before it starts to perform its filtering and sends the results out to the "EUPHORIA" display module.

5.2 Application Pages and the Liaison

Once the application design has been completed using the Mediator, the next stage is to make the application available to end-users. Obviously, it would be very inconvenient to force end-users to individually launch and configure the various modules every time they use the application. Besides being tedious, this approach would force the end-user to be knowledgeable about the details of the application's components and communication structure. Ideally, one would want to provide a user friendly interface that automates application launching. Furthermore, this interface should be easily accessible to users through a publicly available network infrastructure (i.e., local intranets or the Internet).

Applications in our system are made available to end-users through the use of World Wide Web pages called *application pages*. Figure 7 shows an application page for the Medical Image Filtering application described in Section 4.1. Typically, an application page consists of an application description, instructions for using the application, and an embedded Java applet (required) for using the application. Application pages provide an easily accessible, platform-independent interface for locating and using Playground distributed applications over the Internet.



http://www.cs.wustl.edu/cs/playground/release/v3.4.0/index.html

Medical Image Filtering Demo

T. Paul McCartney

A nuclear medicine radioactive blood pool study is used to create movies of the human heart for diagnostic purposes. Each movie consists of a series of images. However, the images suffer from noise caused by ambient radiation, making them difficult to read. This noise can be reduced by digitally filtering the images.

This demo application filters a series of noisy medical images of the human heart, allowing one to view the original and filtered images in an interactive graphical user interface. More information is available on the components of this application.

frame: 1 filename: rvg001.x01

The above pictures show the application's user interface and module configuration. Each of the modules may run on a separate workstation. Note that viewing the module configuration is purely optional; this information is only seen when the user has started the Mediator visual configuration tool.

Instructions

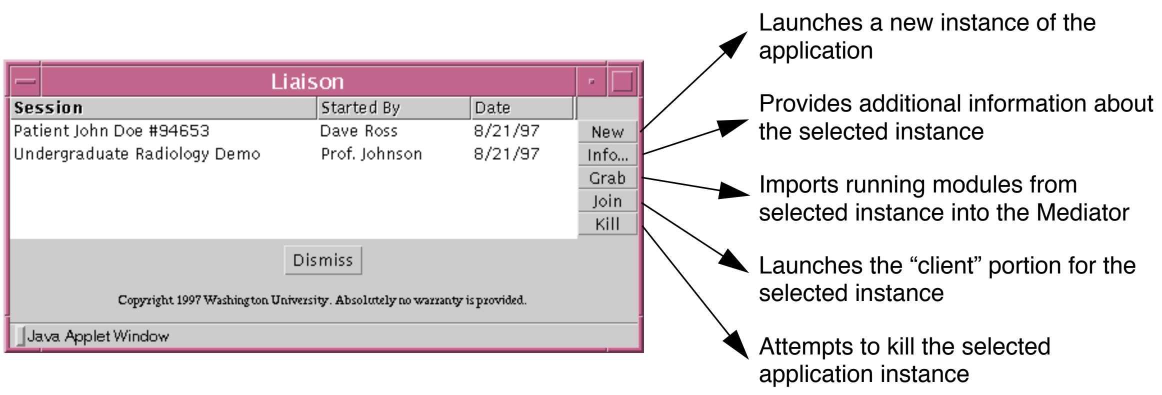
1. Install the Playground User Edition.
2. Start the Application Management System, as described in these online instructions.
3. Press the "Start Liaison" button below. (see these online instructions for using the Liaison)
4. A "Liaison" window should appear. Press the "New" button. After several seconds, the Image Filtering application should be started.
5. After the user interface has been started, edit and change the filename for the image series choice. Possible values are: 'rvg001.x01', 'rvg001.x02', 'rvg001.x03', 'rvg002.x01', 'rvg002.x02', 'rvg002.x03', 'rvg003.x01', 'rvg003.x02', 'rvg003.x03'.

NOTE: *You must use **Netscape Navigator 4.0** or later to run this demo.*

Figure 7: Application Page: Medical Image Filtering

An application page's Liaison applet, shown as the "Start Liaison" button in Figure 7, provides end-users with controls for using the application. When this button is pressed, the user is presented with a window (Figure 8) that contains a list of application instances (i.e., applications already started by the user and possibly others) and several buttons. Users can request that a new application instance should be started by a separately running "Brokerage System" (Section 6) by pressing a button. One may also select a running application instance from the "Session" table and can request information, import the module/connection visualization into a locally running Mediator, or request termination of the application instance. In addition, for applications that are designed using the "client/server" pattern, one may select an application instance representing a "server" and request that a "client" portion should be launched and configured into the server. The Vaudeville video conferencing application (Section 4.2) is an example of a client/server application, where conference participants take on the role of clients in order to join into a shared application. Having the Liaison launch interface on the web allows multi-user applications to be developed fairly easily. Whenever an application server is launched through the Liaison, it is automatically listed in the Liaison window for other participants to join. The implementation of these features is described in Section 6.

One difficulty that we encountered in the Liaison's implementation was with the security issues involved in communicating between the Liaison and the Brokerage System. As described in the next section, portions



Session	Started By	Date
Patient John Doe #94653	Dave Ross	8/21/97
Undergraduate Radiology Demo	Prof. Johnson	8/21/97

Copyright 1997 Washington University. Absolutely no warranty is provided.

Launches a new instance of the application

Provides additional information about the selected instance

Imports running modules from selected instance into the Mediator

Launches the "client" portion for the selected instance

Attempts to kill the selected application instance

Figure 8: Liaison Window

of the Brokerage System typically run locally on the user's workstation. In order to interact with the Brokerage System, the Liaison must access information that is stored locally on the user's file system. In general, Java applets are prevented from reading such information because of security concerns. Our approach to this problem was to use the new security interface classes provided by Netscape Communicator 4 for use in conjunction with Java applets. With these classes, an applet that is digitally signed, through an Internet service such as Verisign [20], can request access to protected information (e.g., the ability to read local files) directly from the user. Another approach to this problem would have been to implement the Liaison as a helper application rather than an Java applet. While this approach would have also solved the security problems, it would have required that a separate Liaison helper application to be built for each supported platform (we support six distinct platforms) and it would have required each user to modify their browser's helper application settings.

6. Brokerage System

The *Brokerage System* is the core of the Application Management System. It is responsible for:

1. Maintaining static information about available modules, workstation performance characteristics, and the application specification/configuration.
2. Maintaining dynamic information about available workstations and running application instances.
3. Performing the resource allocation, launching, configuration, and joining of applications.
4. Executing termination (a.k.a., kill) and import (a.k.a., grab) requests.

The Brokerage System is implemented as a Playground distributed application which consists of three different types of modules: (1) Launchers, (2) Application Daemons, and (3) Brokers. Figure 9 shows a diagram of the Application Management System architecture. The following subsections describe the role of each of these modules.

6.1 Launcher

The Launcher module automates the launching of individual Playground modules on available workstations[3]. The goal of this component was to simplify application development by encapsulating the platform/operating system dependent details of application start-up into this general-purpose reusable

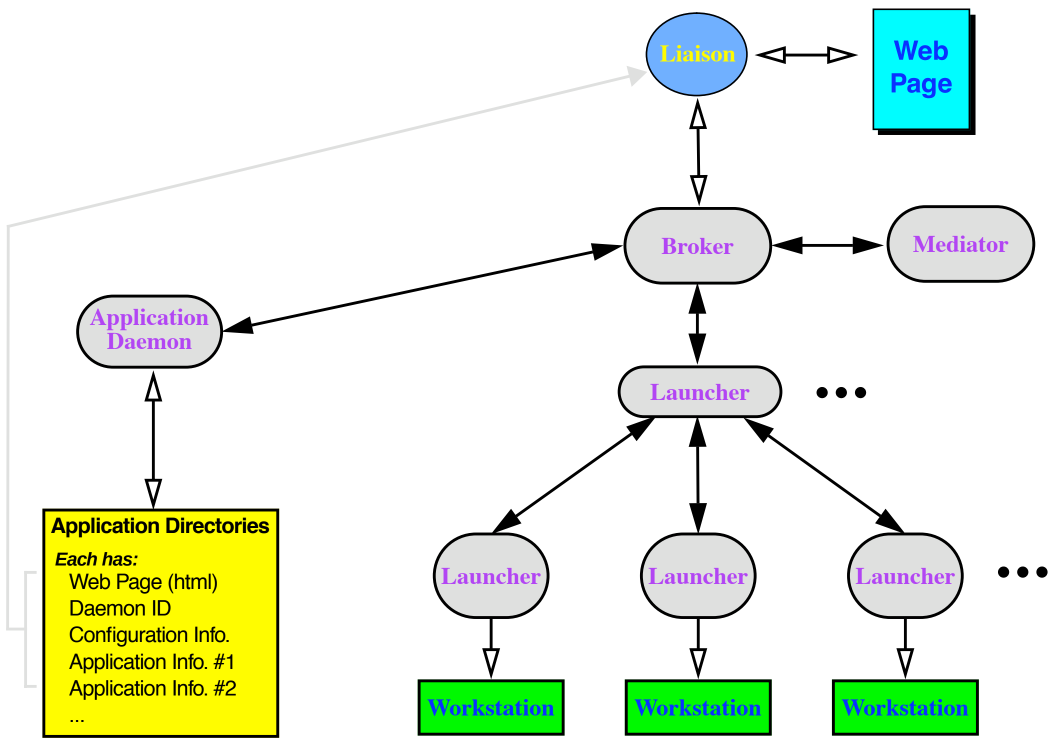


Figure 9: The Application Management System architecture.

module. This approach facilitates the development of Playground applications that tend to be fairly platform independent. Prior to the Launcher's development, each module of a Playground application was launched either manually from the UNIX shell/shell script or programmatically through special purpose operating system calls within the application. For example, the Vaudeville teleconferencing application (Section 4.2) was developed before the Application Management System existed. The server portion was started by executing a shell script; application information was stored in a hard-coded location on the local file system to enable clients to join into a conference. In addition, the source code for the client portion contained operating system calls to fork/exec and join into a conference. While this approach worked, the launching functionality increased the size/complexity of Vaudeville's source code. Only the developers of Vaudeville were capable of successfully launching a conference and joining in clients. Furthermore, it was necessary to modify the operating system dependent code for each new platform that was supported. After the Application Management System was completed, Vaudeville was changed to eliminate the need for scripts and operating system specific launching functionality. Currently, the launching of Vaudeville's modules is automated by the Launcher.

As shown in Figure 9, Launcher modules may be arranged in a hierarchy where each "leaf launcher" operates on a particular workstation and each "parent launcher" manages a number of leaf launchers. This approach allows the system to scale to support a large number workstations. The task of each leaf launcher is to:

3. The term "workstation" is used throughout this paper to represent both workstations, such as Sun's Sparc/Ultrasparc series, and PCs running operating systems such as Windows NT, Linux, and NetBSD.

1. Manage the static information of where each module executable is stored and its launching preferences.
2. Make both static and dynamic workstation performance information available to parent Launchers and/or Brokers.
3. Perform module launching.

Each leaf launcher essentially controls access to its workstation. On the workstation's file system, a directory called the *launch tree* exists which contains information about each module that can be launched on the workstation. For each supported module, this includes optional information such as default command line arguments and environment variables to be passed to a newly launched module. In addition, information about each module must include a mapping from the module's *public name* to the physical location of the module's executable on the local file system. The public name provides a universal way of referring to a particular module that may exist on several different workstations. For example, EUPHORIA [10] is a user interface management system tool that is provided with Playground. EUPHORIA's public name is "EUPHORIA.dpe.cs.wustl.edu" to represent the module name, development group (dpe, for Distributed Programming Environments group), and the Internet domain of the developer. This allows the Application Management System to request the launching of "EUPHORIA.dpe.cs.wustl.edu" on any available workstation rather than having to provide a more detailed launching specification. Leaf launchers read the information in the launch tree and make the list of supported modules (i.e., their public names) available to parent launchers[4] through a Playground published variable.

Leaf launchers also manage information about the workstation's performance characteristics. Prior to the Launcher's execution, a separate benchmarker program is run to evaluate the load-independent performance of the workstation (e.g., MIPS and MFLOPS ratings). Leaf launchers provide this information as well as periodically updated load information to parent launchers through Playground published variables. Parent launchers may then use this information to determine how to allocate modules to be launched to available workstations. Section 6.3 describes the module resource allocation process in more detail.

The Launcher's primary responsibility is to launch modules on its workstation. This action is performed when a parent Launcher or Broker sends a request to a leaf Launcher. A module launch request includes the module's public name, optional command line arguments, and optional environment variable settings. Given such a request, a Launcher launches the module through operating system calls, such as a fork/exec, and obtains an ID called the Playground module communication ID [3]. This ID enables other Playground modules to communicate with the new module, including sending connection and termination requests. When the launching is complete, the ID is passed back to the parent through a Playground published variable.

A parent launcher may manage potentially many other launchers, each of which may be run on separate operating system platforms. For example, one may have launchers running on DEC Alphas, Sun Ultrasparcs, and PCs which are all being managed by a single parent launcher. Since Playground hides the details of interprocess communication from the developer, modules running on these platforms may communicate without any special development effort. The modules of an application that are to be launched may be allocated to any of these (provided that the executables are available) without

4. The term "parent launcher" is used in this discussion to refer to both Launchers and Brokers since launching delegation functionality is common to both.

compatibility concerns, enabling the system to harness the power of a heterogeneous collection of workstations.

6.2 Application Daemon

An *application directory* is a web-accessible directory that contains information that is specific a particular Playground application. As seen in Figure 9, this includes:

1. Application page HTML source and associated documents.
2. Liaison applet byte-code.
3. Configuration specifications for launching new applications or joining existing servers.
4. Application Daemon's Playground module communication ID.
5. Information about each running instance of the application.

The Application Daemon module manages the information contained within a number of application directories. The Application Daemon acts as a shared server that is accessed by the Broker modules (Section 6.3) of various users. Upon start-up, the Application Daemon reads the configuration specification files contained within its designated application directories. The configuration specifications include a list of the application module public names and their communication configuration. These files are generated automatically by the Mediator visual configuration tool, as described in Section 5.1. The information contained within the configuration specifications are made available to other modules through Playground published variables. The Application Daemon also writes its Playground module communication ID to a special file in the application directory.

As described in the next section, the role of the Broker module is to launch and configure a complete application instance. Upon a successful launch, the Broker sends the Application Daemon instance-specific information about the newly launched application. This information includes a listing of Playground module communication IDs for each module in the application, the instance name (optional), the user's name (optional), the activation date, and an optional comment string. The Application Daemon writes this information to a file in the application directory (a separate file is maintained for each instance). Periodically, it "pings" the application to verify that the application is still running. When it detects that the application has terminated, it removes the application instance file from the directory.

Since the application directories are web-accessible, the Application Daemon ID and instance information can be viewed by the Liaison when it is activated. The Liaison uses this information to construct its table of running instances (Figure 8) and to inform the Broker of the Application Daemon's location.

6.3 Broker

The Broker module acts as an agent for starting or joining a complete Playground application. Unlike the Launcher and Application Daemon components, which are typically shared among users, each user runs their own Broker in order to interact with the other components of the Application Management System. As shown in Figure 9, the Broker may communicate with the Mediator, Liaison, Application Daemon, and Launcher components.

The Broker can receive an application launch request from either the Mediator (Section 5.1) or the Liaison (Section 5.2) based on end-user interaction. Requests from the Mediator include a complete specification of the application's modules to be launched. These requests are specified by the user through the Mediator's visual configuration functionality. In contrast, the Liaison sends to the Broker an incomplete application specification (e.g., "start a new instance of application X") and the Application Daemon's ID which is read from the application directory. In response such a request, the Broker obtains the complete application specification by communicating with the Application Daemon.

An application specification consists of a set of modules to launch and a configuration of connections to be made among the modules. Each module launch specification includes the module's public name and workstation performance preferences (e.g., "run this module on the workstation with the lightest load"). Given this information, the task of the Broker is to decide how to allocate modules to available workstations. The choice of how this is accomplished has a substantial impact on the application's performance. For example, Section 4.1 described the Medical Image Filtering application. This application is implemented as a pipeline of several modules that are best run concurrently on separate workstations. In addition, two of the modules (Spatial filters) are more computationally intensive than the others and are best run on the fastest available workstations. The job of the Broker is to take this specification, decide on which each workstation the various modules should be launched, and issue the launch requests to the appropriate launchers that run on those workstations.

As shown in Figure 9, the Broker operates at the root of the launcher hierarchy. Information gathered from the leaf launchers, such as the workstation performance characteristics and the list of available modules, is sent up through the hierarchy to the Broker. This information may be summarized along the way by the hierarchy's parent launchers for efficiency reasons. Parent launchers in the hierarchy perform similar resource allocation tasks to that of the Broker[5], determining how best to allocate modules to the child launchers under their control. Individual module launch requests are propagated down through the hierarchy to leaf launchers. Once launched by a leaf launcher, the module's communication ID is obtained by the launcher and is propagated back up the hierarchy to the Broker. Given the complete set of module communication IDs, the Broker is then able to request the connections to be made among the modules and communicate the set of IDs back to either the Application Daemon or the Mediator.

The option of joining into an already running server application as a client is implemented in a similar way to that of launching a complete application. In this case, the Liaison sends a join request, which includes the server name, to the Broker. The Broker obtains both the application specification of the client and the application instance IDs of the server from the Application Daemon (i.e., from the application instance file stored within the application directory). The launching of the client modules is then delegated to the launcher hierarchy in the usual way. When the launching is complete, the Broker configures the communication among the modules. Connections are requested both among client modules and between the client and the server modules.

7. Conclusion

The Programmers' Playground is a computing environment for developing distributed applications from modular components. This paper presented a visual configuration tool for constructing and debugging Playground distributed applications and an application management system for automating the bookkeeping, resource allocation, launching, and configuration of completed applications through a World Wide Web interface.

5. In fact, the allocation decision functionality is implemented as a common set of classes that both the Broker and Launchers share.

Our hope is that the modular nature of The Programmers' Playground will facilitate the development of a library of general purpose, reusable component modules. The visual configuration tool enables an application developer, who may or may not be skilled at textual programming, to design applications by taking these "off-the-shelf" component modules and interactively configuring the applications. Given a completed application constructed in this way, a separate Application Management System automates application launching and client/server join operations, enabling end-users to use the application.

Using the brokerage system modules to control module launching on arbitrary Internet workstations both simplifies the development of distributed applications (i.e., reducing the amount of operating system specific functionality in the applications) and enables one to harness the power of a heterogeneous army of workstations. One can imagine "farms" of high performance workstations or super computers being made available to paid subscribers over the Internet in this way. That is, a user's application could consist of both modules with low computing requirements which are run locally on the user's own computer and modules with high performance computing requirements (e.g., real-time virtual reality rendering) which are run remotely on more powerful workstations on a pay-per-use basis.

A goal of our work is to minimize the amount of textual programming required in the application development process. Our system enables the application designer to interactively construct the communication structure of an application and to specify the performance requirements of each module. The Brokerage System is completely general purpose and reusable, allowing many types of applications to be automated without the need for special purpose launching and configuration code.

Acknowledgments

This research was supported in part by National Science Foundation grants CCR-91-10029, CCR-94-12711, and ARPA contract DABT63-95-C-0083.

References

[1] Babaoglu, Ozalp, Lorenzo Alvisi, Alessandro Amoroso, and Renzo Davoli. Mapping Parallel Computations on to Distributed Systems in Paralex. In Proc. IEEE CompEuro '91 Conference, Bologna, Italy, May 1991.

[2] Babaoglu, Ozalp, Lorenzo Alvisi, Alessandro Amoroso, Renzo Davoli, and Luigi Alberto Giachini. Run-time Support for Dynamic Load Balancing and Debugging in Paralex. University of Bologna Technical Report UBLCS-92-3, September 1992.

[3] Goldman, Kenneth J., Joe Hoffert, T. Paul McCartney, Jerome Plun, Todd Rodgers. Building Interactive Distributed Applications in C++ with The Programmers Playground. Washington University Department of Computer Science WUCS-97-14, February 1997.

[4] Goldman, Kenneth J., T. Paul McCartney, Ram Sethuraman, Bala Swaminathan. The Programmers' Playground: A Demonstration. In Proceedings of the Third ACM International Multimedia Conference (MM'95), San Francisco, CA, November 1995, pp. 317-318.

[5] Goldman, Kenneth J., Bala Swaminathan, T. Paul McCartney, Michael D. Anderson, and Ram Sethuraman. The Programmers' Playground: I/O Abstraction for User-Configurable Distributed Applications. *IEEE Transactions on Software Engineering*, 21(9):735-746, September 1995.

[6] Hao, Ming C., Alan H. Karp, Milon Mackey, Vineet Singh, and Jane Chien. On-the-Fly Visualization and Debugging of Parallel Programs. Hewlett Packard Corporation, October 1993.

[7] Kunz, Thomas. Abstract Debugging of Distributed Applications. Institut fur Theoretische Informatik Technical Report, December 1993.

[8] Lewontin, Steve and Mary Ellen Zurko. The DCE Web Project: Providing Authorization and Other Distributed Services to the World Wide Web. OSF Research Institute.

[9] McCartney, T. Paul. End-user Construction and Configuration of Distributed Multimedia Applications. D.Sc. Dissertation, Washington University 1996. Also appears as Washington University Department of Computer Science technical report WUCS-96-24, September 1996.

[10] McCartney, T. Paul, Kenneth J. Goldman, and David E. Saff, "EUPHORIA: End-User Construction of Direct Manipulation User Interfaces for Distributed Applications," *Software-Concepts and Tools,* 16(4):147-159, December 1995.

[11] McCartney, T. Paul and Kenneth J. Goldman. Visual Specification of Interprocess and Intraprocess Communication. In *Proceedings of the 10th International Symposium on Visual Languages*, October 1994, pages 80-87.

[12] Neuman, B. Clifford, Santosh Rao. The Prospero Resource Manager: A Scalable Framework for Processor Allocation in Distributed Systems. *Concurrency: Practice and Experience*, 6(4):339-355, June 1994.

[13] Open Software Associates. OpenUI Technical Overview. 20 Trafalgar Square, 5th Floor, Nashua, NH 03063.

[14] Parwatikar, Jyoti K., A. Maynard Engebretson, T. Paul McCartney, John D. Dehart, Kenneth J. Goldman. Vaudeville: A High Performance, Voice Activated Teleconferencing Application. To appear in *Multimedia Tools and Applications*.

[15] Richard, William D., Jerome R. Cox Jr., A. Maynard Engebretson, Jason Fritts, Brian Gottliev and Graig Horn. Production Quality Video Over Broadband Networks: A Description of the System and Two Interactive Applications. *IEEE Journal on Selected Areas in Communications*, 13(5):806-815, June 1995.

[16] Scheifler, Robert W. and Jim Gettys. The X Window System. Technical Report MIT/LCS/TR-368, MIT Laboratory for Computer Science, October 1986.

[17] Shapiro, William M., T. Paul McCartney, E.F. Berkley Shands. The Programmers' Playground Application Management System User Guide. Washington University Department of Computer Science WUCS-97-32, August 1997.

[18] Strom, Robert E., David F. Bacon, Arthur P. Goldberg, Andy Lowry, Bill Silvermann, Daniel Yellin, Jim Russell, and Shaula Yemini. Hermes: Unix User's Guide, Version 0.8alpha. Technical report, IBM T.J. Watson Research Center, Yorktown Heights, New York, USA, March 1992.

[19] Sunderam, V. S. PVM: A Framework for Parallel Distributed Computing. *Concurrency: Practice and Experience*, 2(4):315-339, December 1990.

[20] Verisign, Inc., 1390 Shorebird Way, Mountain View, CA 94043. http://www.verisign.com.

[21] Vinoski, Steve. "CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments," *IEEE Communications Magazine*, February 1997.

[22] Vinoski, Steve. Distributed Object Computing with CORBA. C++ Report, August 1993.