LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

MIT/LCS/TR-648

A DISTRIBUTED PROGRAMMING SYSTEM FOR MEDIA APPLICATIONS

Brent M. Phillips

February, 1995

This document has been made available free of charge via ftp
from the MIT Laboratory for Computer Science.

545 TECHNOLOGY SQUARE; CAMBRIDGE, MASSACHUSETTS 02139 (617) 253-5851

A Distributed Programming System for Media Applications

Brent M. Phillips

Telemedia Networks and Systems Group
Laboratory for Computer Science
Massachusetts Institute of Technology

Abstract

This thesis describes the design and implementation of VuDP, a programming system for distributed media applications. VuDP has been implemented as a set of extensions to the VuSystem, a programming environment for media-based applications. VuDP supports distributed media applications in which VuSystem modules can be created and placed on any node in a network. VuDP is integrated with the VuSystem runtime programming environment and dynamic reconfiguration capabilities.

VuDP has been used to develop several VuSystem applications that would have been difficult or impossible to implement without the VuDP extensions. The VuDP remote evaluation mechanism allows for the creation and manipulation of remote interpreters and for setting up both in-band (media flow) and out-of-band (control and event handling) connections between local and remote interpreters. Further, VuDP RPC mechanisms complement remote evaluation by providing tools and services that support the implementation of interactive multi-user applications using the VuDP daemon.

©Massachusetts Institute of Technology 1995

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Overview

In the last several years there has been a rise in the use of loosely-coupled networks of computers, where sets of workstations and/or personal computers are hooked together via an ethernet or other local area network technology. At the same time, the increasing power of workstations and personal computers has allowed for the development of applications that can capture, process, transfer, and output media in different manners. However, writing distributed programs using only the tools provided by most programming languages and operating systems is generally too complex and unwieldy to be worthwhile.

It seems natural, then, to develop a distributed programming system for media-based applications that provides enough support to make distribution both simple and powerful. This thesis discusses a system that allows programmers to write media-based applications whose processing modules are distributed among workstations connected by a local area network. This was done by extending the VuSystem [14][13], a video toolkit developed in the Telemedia, Networks, and Systems group at the MIT Laboratory for Computer Science.

Major design issues in extending the VuSystem to allow for distributed programming were finding the correct users' and programmers' view of distributed programs and supplying the proper tools for writing distributed programs. Distributed programming can involve either explicit distribution by the programmer and/or transparent distribution by the underlying system. Either (or both) may be appropriate, depending on the goals of the system. Further, in an explicitly distributed program each site must provide an execution environment for its portion of the program. Hence, peer execution environments may either be completely separate or have a well-defined shared state and set of interactions. For VuSystem distributed programming, these decisions are influenced by the kind of support VuSystem programmers want for writing distributed applications, as well as constraints on what is possible and practical to do with the VuSystem.

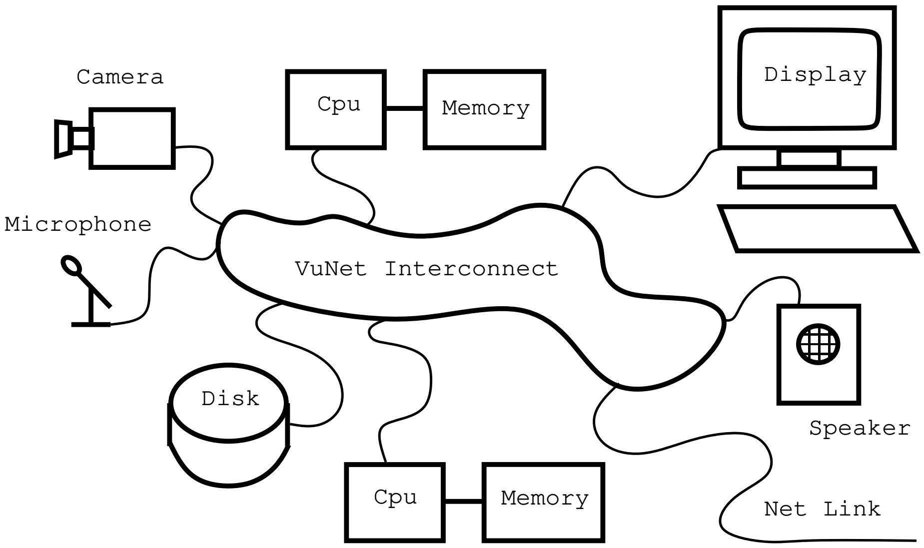

Figure 1.1: The ViewStation

1.2 Media Programming and the VuSystem

Video has become a practical digital media to be captured, stored, manipulated, and played back by computers. However, existing media programming systems only provide limited support for distributed programming. Applications typically run in a single execution environment, and remote resources can only be accessed via mechanisms such as NFS (for remote files) and X Windows (for remote displays).

The VuSystem [14][13] is a toolkit for creating applications that manipulate time-based media, typically digital video, digital audio, and captioned text. A VuSystem application is modeled as a series of reusable modules which create, transfer, and process the media. Example modules are modules that act as video sources, modules that process or filter video, and modules that display video to the user. Two features that distinguish the VuSystem from other video programming toolkits are that the VuSystem is designed to create computer participative applications (where the computer directly manipulates the media) and that the VuSystem is designed to work with live video sources. The ViewStation[24] project integrates the VuSystem with the VuNet, a high speed ATM-based local area network developed within the TNS group. It includes several modern workstations, high-speed links between nodes, and video capture cards known as Vidboards[2].

The research described here extends the VuSystem to allow for distributed programming. In the traditional VuSystem, program distribution is limited to the use of NFS, X Windows, and Vidboards (which may be accessed by any machine on the VuNet). All application-specific modules execute in a common address space. Hence, media sources and sinks that are not directly available to the local machine can not be used by any applications running on that machine. Further, two applications on different hosts that require communication for collaborative processing [1] must both know when and where the communication will occur. Explicitly setting up interactive sessions is awkward and greatly reduces the utility of multi-user applications.

1.3 The VuSystem Programming Environment

The VuSystem programming model is based on the idea that media applications can be divided into sections that do "in-band" processing and sections that do "out-of-band" processing. This idea is illustrated in figure 1.2. In-band processing refers to processing which touches the underlying media stream; for video, this corresponds to directly manipulating video frames. In-band processing is performed on a video or audio stream, and so the performance of the in-band processing is critical, especially when live video is being manipulated in real time. Out-of-band processing refers to creation and configuration of in-band modules, event handling, and other high-level programming concerns. In-band modules are implemented in the VuSystem as a library of C++ modules for efficiency. (Less time-critical modules may be implemented either in C++ or as a library of Tcl routines.) The C++ class system allows for quick and simple creation of new modules that subclass existing modules and, when necessary, for creation of new modules from scratch. Because in-band and out-of-band processing have very different characteristics, separating them allows their architectures to be optimized independently.

VuSystem applications are written as out-of-band Tcl [17] scripts which typically create and configure a variety of modules and then dedicate themselves to control message passing where they decode user events, initiate the appropriate actions, and display feedback to the user. Shown in figure 1.2 is the architecture of a simple application consisting of a video source, a video filter, and a video sink. The out-of-band application code - a Tcl script - sets up a video source module, a filter module, and a sink module, and connects the inputs and outputs of the modules so that the media flows from the source through the filter and into the sink. In-band code inside the three modules handles direct manipulation of the video. The out-of-band code communicates with the in-band code through commands and callbacks.

Because Tcl is an interpreted language and VuSystem applications are written as

[1] An example of such an application is VsTalk, a VuSystem application that allows two users to view live video of each other. In the traditional VuSystem, a VsTalk session must be prearranged via email, a phone call, or face to face communication.

Tcl scripts, great flexibility is provided in reconfiguring applications at run-time. The VuSystem has a run-time visual and textual programming environment that allows a user to quickly and easily reconfigure an existing program either by entering Tcl commands to be interpreted or by simply moving around graphic representations of modules [25]. This makes the VuSystem a programming environment well suited to the creation, debugging, and testing of media applications.

There are several reasons why the VuSystem is an appropriate platform for this research. First, it is a working programming system which, when coupled with its companion, the VuNet [3], a high speed network, is well suited for the development of media-based applications. Second, it is interesting to write distributed programs in the VuSystem because of its interpretive textual and graphical programming environment. And, finally, the traditional VuSystem does not allow for generic distributed programming.

Distributed programming is a natural extension that greatly increases the power and flexibility of VuSystem applications without requiring extensive modifications to the VuSystem infrastructure. In short, the distributed programming extensions to the VuSystem demonstrate the viability of a dynamically reconfigurable distributed multi-layer media programming system.

1.4 Related Work in Distributed Programming Systems

Writing programs that run simultaneously on different machines is a difficult and time-consuming process because of problems with asynchronous communication, partial failures, data marshalling, and other distributed programming concerns. This is unfortunate, since there are opportunities to realize increased fault tolerance, performance, power, and flexibility by leveraging the resources of several computers connected to a local area network.

Related work on distributed programming can be categorized by both the goals of supporting distribution and by the scope of the support for distributed programming. There are three basic reasons for wanting to write distributed programs: to improve fault tolerance, to increase concurrency to improve performance, and to access special resources that are only available from certain machines. Distributed programming facilities can be designed for one, two, or all three of these reasons. Further, there are different types of interaction between remote processes. One type is the traditional RPC interface implemented by message passing, where a server exports a standard set of services that remote clients can access. A second type is remote evaluation, where a server acts as a sort of soft programmable abstraction to which a client can send both the program for the server to execute and the data for it to operate on. Finally, support for distributed programming can either be in the form of general operating system

facilities for distributed programming or extensions to a specific language to ease writing distributed programs in that language. The Client-Shell Distributed Architecture [20], the ISIS[6] project, Mach [1], and Chorus [19] provide general operating system level support for distributed programming in any language, while the rest of the related work focuses on either creating a new distributed programming language or extending an existing language to facilitate distributed programming.
At a low level, UNIX provides system calls which allow programs to send and receive packets over a local area network using a variety of communication protocols. Berkeley UNIX provides a socket[8] abstraction for handling this style of point-to-point intermachine communication. However, this built-in support for point-to-point communications is often too low-level for everyday use in applications development. Little or no direct support is provided for data marshalling, synchronous communication, handling partial failures, and other concerns. Some form of additional support is needed for distributed programming to be practical.

1.4.1 The Client-Shell Distributed System Architecture

The Client-Shell Distributed System Architecture [20](also known as PPM for "Personal Program Manager") adds an extra layer between the command shell and the operating system. PPM takes the view that distributed programs typically involve many different communicating processes running on different machines, and provides services to facilitate location-transparent creation and control of structures involving these distributed processes. PPM provides services for creation and execution of jobs and processes (including creation and I/O configuration of remote processes), bookkeeping (to allow a user to obtain information about all his active jobs and their component processes), control (to allow a user to suspend, resume, or terminate related jobs and processes as a unit), and information sharing (to allow different jobs and processes to communicate through a shared name space). This allows users to easily set up and manipulate distributed programs, even if the component processes are running on different machines.

1.4.2 ISIS

ISIS[6] is a system for fault-tolerant distributed computing. The focus of ISIS is to use distribution to provide fault-tolerance, since a program or data object that exists at several sites can use its extra copies as backups that can take over in the case of a network or machine failure. This is done through an abstraction known as a resilient object, which is an object that is automatically replicated at several sites to provide fault-tolerance. In ISIS, control of distributed computing is wrapped into the resilient object abstraction. Resilient objects are created using a special programming language which (in the current implementation) is an extension of C. ISIS supports a global naming mechanism so that conventional applications can easily find and interface with resilient objects using an RPC-like mechanism. Important features of the implementation of resilient objects are the multicast communication primitives built on top of the standard UNIX point-to-point facilities and the use of transactions and locking mechanisms to handle errors. Using ISIS, fault-tolerant software can be written by first writing and debugging an application entirely in a conventional programming language, reimplementing the critical sections as resilient objects, and interfacing the resilient objects with the conventional sections of the application.

1.4.3 Mach

Mach[1] is a multiprocessing operating system kernel and environment meant to be used as a foundation for UNIX development. Most of Mach's main features (such as support for both tightly-coupled and loosely-coupled multiprocessors, the separation of the UNIX process abstraction in tasks and threads, a virtual memory system, and kernel debugging facilities) are not directly relevant to providing support for distributed programming. However, Mach does provide a capability-based interprocess communication facility that transparently extends across network boundaries. This facility can be used as an alternative to internet domain sockets for interprocess communication in distributed programs, and has the advantage of providing location independence, data typing, and improved security. To provide this new form of interprocess communication, Mach supports the port as a basic transport abstraction. Ports are objects which represent services, and are accessed much like objects in an object-oriented system - a command is sent to an object requesting that it perform some service. Ports are protected by the kernel, and processes must have the proper capabilities in order to access other processes' ports. Note that the Mach kernel does not directly provide any support for communication over a network. Instead, processes known as network servers direct inter-node messages to the appropriate destination through network ports. Network ports are ports to which processes on multiple hosts have access rights, so that a message sent to a network port is actually sent over the network to the host running the process that owns the network port.

1.4.4 Chorus

Similar to Mach, Chorus[19] is a nucleus upon which distributed operating systems may be built. Chorus is not built on top of a specific sub-system; instead, it provides generic tools designed to support various host systems that can co-exist on top of the Chorus nucleus. Traditional operating systems can be built on top of Chorus by creating servers built on top of the Chorus nucleus that offer higher-level services, and then combining the servers into a sub-system that provides a traditional operating system interface. The Chorus nucleus manages the physical resources of each machine, provides support for the actor, thread, and port abstractions, and supports location-transparent inter-process communication. A Chorus actor is a set of resources (address space, threads, etc.) similar to a Unix process or a Mach task. Chorus threads are units of sequential execution that run within actors, much like Mach threads run within Mach tasks. Chorus ports are location-transparent entities attached to actors. The most important support that Chorus provides for distributed programming is the synchronous and asynchronous IPC facilities of Chorus ports. All inter-thread communication is done by sending messages to ports, and communication is location-independent, so that two threads within the same process communicate via ports in the same way as two threads in separate processes on separate hosts. Within a single machine, the Chorus nucleus handles inter-actor communication; between machines, the nuclei and a network server (built on top of the nucleus) work to route messages to the appropriate machine so they can be delivered to the appropriate port.

1.4.5 Argus

Argus [15] is a programming language and system designed to support distributed implementation of systems that must maintain on-line state. Argus supports both distributed programming and transactions through mechanisms known as guardians and actions. Argus guardians are objects that encapsulate and protect a set of resources while providing access to the resources through a set of handlers. Handlers are special procedures that can be used to access the resources of the guardian in a controlled fashion. Guardians maintain internal state and keep vital information in stable storage so that they can be rebuilt after crashes. Conceptually, guardians and handlers are much like objects and methods in object-oriented programming languages. Next, computations in Argus may be encapsulated within "actions" which (like transactions) are both serializable and total. By providing guardians and actions, Argus eases the job of writing reliable distributed programs that deal with partial failures and concurrency.

1.4.6 Medusa

Medusa [27] is a system that uses a peer to peer architecture to create, control, and configure networked media devices. All Medusa entities are active modules which may represent applications as well as lower-level modules such as cameras and displays. Medusa focuses on providing secure, reliable connections between modules in an environment where all modules are peers. Medusa is similar to the VuSystem in many respects, though two notable differences are the VuSystem's in-band and out-of-band hierarchy and interactive programming environment. Medusa grew out of the Pandora project described in [11].

1.4.7 Hermes

Hermes [9] is a high level language in which it is simple to start up new processes and connect them together to allow distribution of different parts of a program. Hermes provides high-level communication facilities built on top of the underlying mechanisms to allow Hermes programs to be easily portable to different kinds of systems. To this end, the Hermes compiler hides all the details of the low-level IPC used to implement the communication mechanisms. This facilitates breaking programs into separate communicating pieces that may be easily distributed over a network without getting the programmer bogged down in the details of low-level IPC mechanisms.[2]

1.4.8 Concert/C

Concert/C [4] is an extension to the C language to provide improved process management, new data types, and communication mechanisms to ease IPC among C programs. Concert/C provides primitives for interconnecting processes, for passing bindings between processes, and for exporting bindings to shared files so that C programs running on different machines can be set up in a client/server architecture using RPC for communication. Both synchronous and asynchronous point-to-point communication mechanisms (which hide the details of the underlying IPC mechanism) are provided. Hermes and Concert/C provide a similar set of utilities for network programming.

1.4.9 Avalon/C++

Avalon/C++ [26] is a set of extensions to the C++ language that provide support for developing fault-tolerant software via the use of transactions and modules with recoverable state. Avalon/C++ focuses on providing tools to use distributed programming for increased fault tolerance. Primitive classes of recoverable data are introduced which may be used to build up more complex recoverable data types, and a transaction mechanism is provided to support fault-tolerant programming. Avalon/C++ encourages writing applications in a client/server fashion, and supports server classes which can export certain methods for use by other processes via an RPC-like mechanism.

1.4.10 Avalon/LISP

Avalon/LISP [7] contrasts with Avalon/C++ in that it is a set of extensions to LISP to provide support for general remote evaluation. In Avalon/LISP, evaluators are first class objects, and can be passed in messages from clients to servers. This gives fine-grain control over where evaluation is done for different

[2] Hermes grew out of the NIL programming language[18], and so NIL will not be discussed here.

parts of a program; some parts can be evaluated locally, and others remotely. No support is provided for transactions or other locking mechanisms; the focus of Avalon/LISP is to allow for general remote evaluation.

1.4.11 REV

REV [21] is a set of extensions to CLU to provide transactions and support general remote evaluation. Servers are not viewed as providing a fixed set of services, but are rather seen as programmable devices that may be sent both code to execute and data to execute it on. This adds a great deal of flexibility to the construction of distributed programs, but also raises security concerns. REV also provides a transaction mechanism that is used to deal with partial failures.

1.4.12 Distributed ML

Distributed ML (DML) [12] provides distributed programming extensions to Concurrent ML, which is itself a version of Standard ML enhanced to support concurrent programming. DML provides a data object called a port group, which is a fault informative asynchronous multicast channel that can be used as the basis for writing distributed programs.

1.4.13 Marionette

Marionette [23] is a language designed for distributing computation over a network for parallelism and increased performance. Marionette uses a master-slave model for parallel computing, where a "master" process sends work to "slaves" that are transparently distributed over a network.

1.4.14 PROFIT

PROFIT [10] is a distributed object-based language designed for financial analysis where distributed processes must share primitive objects. In PROFIT, processes running anywhere over a network can be sharing objects. A main feature of PROFIT is that it allows dynamic reconfiguration of the primitive objects processes are using and sharing.

1.4.15 Durra

Durra [5] is an application-level programming language whose philosophy is similar to the VuSystem in that Durra programs connect and configure component modules which may be implemented in any language. Like the VuSystem, Durra separates concerns of component development from the use of those components. Unlike the VuSystem, Durra is meant as a general purpose application-level language, and is not specifically designed for writing media applications.

1.4.16 The Multicast Backbone

The Multicast Backbone, or Mbone[16], is a virtual network on the Internet used for multicast of audio and video. The Mbone uses IP multicast addressing and special routers that handle multicast addressing to support the multicast of audio and video streams over wide areas using the same physical layer as existing point-to-point networks. A suite of Mbone video conferencing applications has been developed, including sd (the session directory tool for active video conferences), vat (the Mbone audio tool), vic (the Mbone video tool), and wb (the shared white board.)[3]

1.4.17 Summary

To summarize, there is a significant amount of work on languages designed for or modified to accommodate distributed programming and on systems that use distribution for improved fault tolerance. The VuSystem distributed programming extensions described in this thesis have several novel aspects (described in the next section) not covered in the research described above.

1.5 VuSystem Distributed Programming

VuSystem Distributed Programming (known as VuDP) is an extension to the VuSystem programming toolkit. The goal of VuDP is to allow for the distribution of a VuSystem application over several hosts in a network. This allows applications to access special resources attached to certain workstations, allows distribution of computationally intensive work across several machines, and facilitates collaborative processing between processes in different address spaces or on different machines. VuDP focuses on providing practical, usable programming tools without requiring major modifications to the VuSystem software. This means working within the limitations of the current VuSystem, such as the constraint that all applications are single-threaded. Finally, because VuDP applications have access to the VuNet[3](a high-speed local area network), the high-bandwidth communication capabilities of the VuNet influence the VuDP design and implementation.

VuDP hides most of the details of low-level IPC, detection and handling of partial failures, security, and integration of distributed programming with the VuSystem run-time programming environment. The VuDP implementation is modular and required only a few minor changes to the traditional VuSystem infrastructure. VuDP, like the rest of the VuSystem, assumes the existence of a network file system (such as NFS) and a network windowing system (such as X Windows). VuDP strives to provide a flexible, powerful distributed media programming

[3]These tools are largely undocumented in the literature, but may be found by anonymous ftp to *ftp.ee.lbl.gov*.

system while consciously avoiding becoming more complex or more general than it really needs to be.

1.5.1 Novel Aspects

VuDP has several characteristics that set it apart from existing distributed programming languages and distributed programming systems. Most related work focuses on providing either broad system-level support for distributed programming or on extending or creating a specific programming language that provides tools to simplify distributed programming. VuDP, on the other hand, is a set of extensions to a multi-layer programming system that is specifically designed for media computation. The most important novel aspects of VuDP are that it:

- is designed for creating media-based applications that run over a gigabit local area network
- has an interactive textual and graphical user interface to allow dynamic program reconfiguration at runtime
- blends RPC and remote evaluation functionality to meet the needs of media applications

1.5.2 Research Issues

Several research issues are addressed in the design and implementation of VuDP. Some issues - such as providing the proper communication primitives and handling partial failures and security problems - are present in any distributed programming system. Other issues - such as the integration of distributed media programming with the VuSystem dynamic reconfiguration capabilities - are consequences of the novel aspects of VuDP. The most important issues in the design and implementation of VuDP are:

- what the remote programming interface should look like
- what communications primitives should be supported
- how to integrate the remote programming interface with the VuSystem runtime programming environment
- how to integrate remote modules with the VuSystem dynamic configuration capabilities
- how to handle the security problems associated with remote evaluation
- how to handle partial failures

1.5.3 Organization

Chapter Two provides background material on the VuSystem and goes into more detail on the services provided by VuDP and the ways that VuSystem programmers can use VuDP to write distributed VuSystem applications. Chapters Three and Four treat the design, use, and implementation of the in-band partition and out-of-band partitions of VuDP, respectively. Chapter Five presents several different applications of VuDP, including remote evaluation, the VuDP daemon, and several distributed VuSystem applications. Chapter Six presents the results of VuDP performance testing. Chapter Seven concludes the thesis.

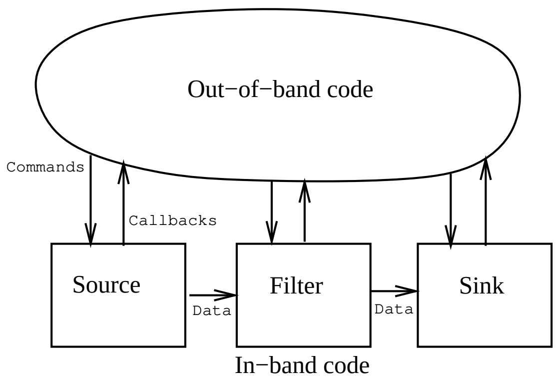

Figure 1.2: The structure of a simple VuSystem application

Chapter 2

VuDP: VuSystem Distributed Programming

VuSystem applications are created by configuring and connecting modules that operate on a media pipeline. Media data flows through the in-band modules and can be directly manipulated by each module. In a traditional VuSystem application, all modules must exist in the same local environment on a single host machine. Hence, in the context of the VuSystem distributed programming means creating, configuring, and manipulating modules at remote sites. Using VuDP, modules may be placed in remote execution environments distributed across several hosts.
A distributed VuSystem application must have mechanisms for both in-band (media flow) and out-of-band (control message) communication between peer execution environments. These remote communication mechanisms are encapsulated within the basic tools that VuDP provides for constructing distributed VuSystem applications: remote sources, remote filters, and remote sinks. By using these VuDP facilities, applications can create, configure, and connect modules and place them on any host in the network. In-band media flows and out-of-band control messages are seamlessly routed to the appropriate module in the appropriate address space. Hence, remote modules may be manipulated and connected with other modules in the media stream just like local VuSystem modules.

2.1 Approach

VuDP services are built on top of the basic VuDP transport services provided by the VsTcp and VsNet modules. The basic architecture of VuDP is illustrated in figure 2.1.
The VsTcp and VsNet modules provide network-based communication channels. Given two execution environments, VsTcp modules can be used to establish a

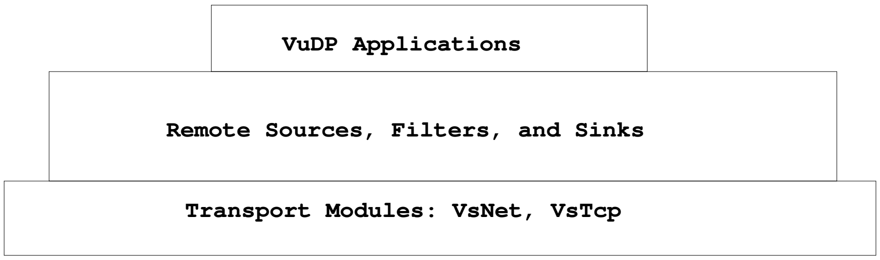

Figure 2.1: VuDP Approach

TCP connection for in-band data. Similarly, VsNet modules can be used to set up a TCP connection for out-of-band data, allowing remote execution environments to exchange control messages over the network.

2.1.1 Motivation

There are many reasons why VuSystem applications may create and configure remote modules. One motivation for distributing program modules is to allow placement of a computationally intensive filtering operation on an idle machine in order to improve performance. Another reason is to allow the creation of a remote source module that can access a special media source (such as a camera) that is only available to a certain host. An example of this is shown in figure 2.2. Without the VuDP remote sourcing capability, application A would not be able to access camera Z. With VuDP, the camera is accessible, as illustrated in figure 2.3.

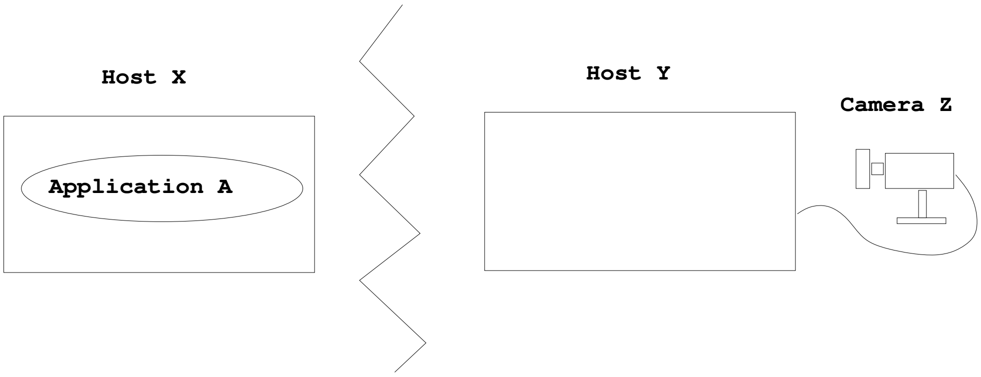

Figure 2.2: Without VuDP

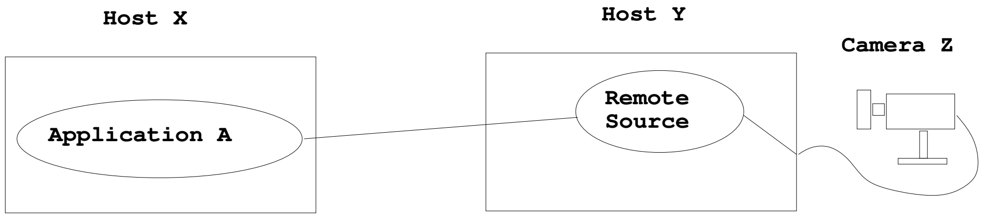

Figure 2.3: With VuDP

2.1.2 Fault Tolerance

VuDP does not need a complex transaction mechanism to provide an appropriate level of fault-tolerance. For explicitly distributed modules, all VuDP provides (and all it really needs to provide) is a reliable failure reporting mechanism which informs processes with remote connections when there is any sort of failure that breaks the connection (such as buggy software, network failures, and remote machine failures). For transparently distributed modules, failure messages are relayed to the local environment so that failures of transparently distributed remote modules appear the same as failures of local modules. A complicated automatic failure recovery system is not necessary; simple mechanisms which relay remote failure messages to the local interpreter for display to the user are sufficient. Transactions are not necessary because, for VuSystem applications, there is generally no way to mask failures through replication. For example, if a video source attached to only one host is being remotely accessed and the host crashes, there will be no way to access that video source again until the host recovers. Transactions would only help in these situations if it were necessary to enforce some sort of atomicity constraints on the actions of remote objects, and such constraints are not necessary for the present VuSystem applications.

2.1.3 Security

Security is an issue whenever distributed programming capabilities might allow an application to have access to resources that it should otherwise be denied. Authentication ensures that all VuDP remote processing runs with the privileges of the user running the application. Thus, a user can do no more with VuDP than he/she could by logging into a remote machine.

2.2 The VuSystem

Before delving into the intricacies of using VuDP, it is useful to examine traditional VuSystem applications in more detail. VuSystem applications are programmed in Tcl, an interpretive scripting language. Application scripts are the "out-of-band" code that creates and controls "in-band" modules. A VuSystem application script is run by creating a Tcl interpreter, initializing it to run a VuSystem application, and then having the interpreter interpret the application script. Each VuSystem application is run by one interpreter in one execution environment with one name space, and all applications are single-threaded. Because the only data type in Tcl is strings, objects which are too complex to convert to and from strings are represented by object commands. For each such object, a new Tcl command is registered with the interpreter. Operations on objects are invoked by the object command, with the first argument typically specifying the operation and the other arguments specifying the arguments for the operation. For more details on Tcl and VuSystem programming, see [14].

2.2.1 The VsPuzzle Application

VsPuzzle is an example of a traditional VuSystem application. VsPuzzle opens a video source and displays a scrambled image of the video in a window; the user can then attempt to unscramble the image.

When run, the VsPuzzle script creates a video source module, a puzzle filter module, and a screen video sink module. The output of the video source is connected to the input of the puzzle filter, and the output of the puzzle filter is connected to the input of the screen sink. The application then "starts" the modules, which initializes in-band processing and starts the media flow. The media then begins to flow from the source through the filter to the sink, and the script dedicates itself to event processing.[1]

Note the separation of the application into in-band and out-of-band sections, as illustrated in figure 2.4. The out-of-band code communicates with the in-band sections via commands, and the in-band sections can return results and execute callbacks to communicate with the out-of-band sections.

In the traditional VsPuzzle application all modules exist on the local machine. The media flow runs only between modules in the same address space on the same local host machine. X-Windows may be used to place the output display on a screen different than the one physically attached to the host running the application, and NFS may be used to access video file sources present on remote machines, but in general all component modules of the application must be local.

2.2.2 Creating, Starting, Stopping, and Destroying

It is important to clarify the semantics of "create", "destroy", "start", and "stop" in the context of VuSystem modules.[2] To create a module means to bring it into

[1] Event processing mostly involves handling user input events.

[2] Because all VuSystem modules are represented as objects, starting and stopping an object means invoking its "Start" and "Stop" methods.

existence; consequently, to destroy a module means to irrevocably remove it from an application. Once created, VuSystem modules which do in-band processing need to be started in order to initialize and begin in-band processing. Once started, a module can be stopped in order to halt in-band processing without destroying the module. A module that does media processing must normally be stopped in order to reconfigure it, though usually this stopping and restarting is performed transparently by the reconfiguration code.

2.2.3 VuSystem Interactive Programming

One interesting aspect of VuDP is how it smoothly integrates distributed programming with the VuSystem dynamic configuration capabilities, consisting of VuSystem control panel mechanism and the VuSystem interactive programming environment.

The Control Panel

In the VuSystem, most applications have a "control panel" which can be used to dynamically reconfigure module options. For example, any application which has a video source module will have control panel options for configuring video source options such as black and white or color, 24-bit color or 8 bit color, etc. VuDP supports this control panel mechanism and allows for the control panel to configure remote sources, filters, and sinks in the same way the traditional VuSystem control panel is used to configure local sources, filters, and sinks. An example of a VuSystem control panel controlling one audio source and one video source is presented in figure 2.5.

Interactive Programming

The VuSystem provides a run-time interactive programming environment where modules can be dynamically created and deleted. Every VuSystem application has a "Program" button which, when selected, opens the interactive programming environment. The interactive programming environment consists of two windows. The top window contains a graphical representation of the application, and the bottom window is a textual interface to the Tcl interpreter.

The upper window contains a graphical representation of the in-band modules of the application and illustrates the flow of media through the modules. Graphical programming tools are provided which allow users to dynamically create, delete, disconnect, and reconnect application modules.

The bottom window allows a user to enter Tcl commands directly into the interpreter as the application is running. Any command which can be included in a VuSystem application script can be entered into the interpreter in the interactive programming environment.

An example of the VuSystem visual programming windows is illustrated in figure 2.6.

2.2.4 The VsEntity Class and VuSystem I/O Support

The VuSystem uses the C++ class hierarchy for its modules, which allows for classes to be organized in a hierarchy and for related classes to inherit functionality from their ancestor classes so that duplication of code is avoided. Most VuSystem modules are associated with a C++ class. The state associated with a module is implemented by instance variables, and module functions are implemented as class methods. For example, there is a VsNetClient class which implements VsNetClient modules. The various creation, starting, and stopping routines for the VsNetClient modules are methods for the VsNetClient class.
An especially important VuSystem class is the VsEntity. Most VuSystem modules (including all sources, sinks, filters, and all VuDP modules) are subclasses of the VsEntity class.

Input and Output Callbacks

One useful feature provided by the VsEntity class is the ability to export callbacks for input and output. The VsEntity class allows objects to specify an "Input" method and link the input method to a file descriptor such that the input routine is automatically executed when new input is available on that file descriptor. Hence, modules which maintain open file descriptors (or network sockets represented by file descriptors) can be set up so that they are automatically notified (via the execution of the "Input" method) when new input is available. Similarly, any object which is of the VsEntity class (or some VsEntity subclass) can provide an "Output" method. Then, when the module is ready to produce new output, a callback will be set up to perform the "Output" routine in the background. This allows modules to set up an output call without having to block the application until the output operation has been completed.

Tcl Callbacks

One of the most important features of the VsEntity is the ability to specify and execute Tcl callbacks. Recall that VuSystem applications are Tcl scripts which create, configure, and control modules that perform the actual media manipulation. Modules are Tcl "objects" represented as new Tcl commands, and module procedures may be called by invoking the Tcl command for that object (typically the object's name) and passing as parameters the name of the module method and any arguments. In-band VuSystem modules are written in C++ and can not directly execute any Tcl code or communicate with their controlling script except when returning results of commands. To allow the C++ modules to communicate with the Tcl script, the VsEntity class allows a module to asynchronously execute a section of Tcl code in the environment of its parent Tcl interpreter.

To implement this, the VsEntity class has a public “callback” instance variable which can contain any Tcl command. (Note that all Tcl commands and simple data are simple strings.) Any VsEntity method (or friend) can execute the callback at any time via the VsEntity EvalCallback method. Note, though, that when a callback is performed with EvalCallback it may not be executed immediately - callbacks are actually performed in the background. This avoids blocking the execution of C++ code to perform a Tcl callback, but causes other problems such as the arbitrary reordering of callbacks.

One use of the callback is to allow the low-level modules to inform their controlling application of important events. For example, when a VsNetListener module (where the VsNetListener class is a subclass of VsEntity) establishes a connection and instantiates a new VsNetServer module to handle the connection, it informs its controlling application via a callback.

Starting and Stopping

The VsEntity class also provides “start” and “stop” methods, which provide code that needs to be executed at the beginning of and at the end of in-band processing, respectively. As an example, imagine a VuSystem application that wants to use a file as a source of media. Before the media can begin to flow, the file must be opened in the appropriate fashion; this would be done in the “start” method for the file source. When the application is finished with the file source, the “stop” method closes the file.

When the “start” method of any subclass of VsEntity is called, the “start” method of that class, and of any superclasses up to and including the “start” method for the VsEntity class itself, are automatically executed. Similarly, when the “stop” method of any subclass of VsEntity is called, the “stop” method of that class, and of any superclasses up to and included the “stop” method for the VsEntity class itself, are executed. As an example, recall the VsPuzzle module, which is a subclass of VsFilter, which is a subclass of VsEntity. When a VsPuzzle module is started, the “start” methods for VsPuzzle, VsFilter, and VsEntity are all executed. This assures that all set-up for in-band processing at the puzzle, filter, and entity levels is performed before the media flow actually begins.

Note that in the VuSystem starting or stopping an object also recursively starts or stops all of its children. This is generally useful, as a single high-level “start” command can be used to start all the modules in an application.

Object Commands

Module commands are normally implemented as "friend" procedures of the appropriate class. Because objects commands are implemented as new Tcl commands, they can't be class methods, but instead must be friends of the class.

2.2.5 VuDP Modules

The VuDP implementation takes advantage of the multi-layered structure of the VuSystem programming environment. The implementation of the basic VuDP tools (remote sources, sinks, and filters and remote evaluation) consists of a few new C++ modules and several Tcl scripts; very few changes to existing VuSystem modules or infrastructure were necessary. The fact that nearly all of the VuDP implementation is contained within a few new modules and scripts is evidence that distributed programming (as provided by VuDP) is a natural extension to the VuSystem.

The implementation of VuDP is based on seven modules: VsNetClient, VsNetServer, VsNetListener, VsTcpClient, VsTcpServer, VsTcpListener, and VsRemote. All of these modules are low-level VuSystem modules, implemented in C++. The VsTcp and VsNet modules are used for creating network based in-band and out-of-band communication channels, respectively, and are available for general use by any VuSystem application. The VsRemote module is a special module used to implement the transparent distribution of remote sources, remote sinks, and remote filters.

All VuDP modules (VsNetClient, VsNetListener, VsNetServer, VsTcpClient, VsTcpListener, VsTcpServer, and VsRemote) are implemented by C++ classes which are subclasses of the VsEntity class. The state of a module is maintained by the instance variables of the appropriate class, and module procedures are implemented as class methods. The way that the VuDP modules fit into the VuSystem hierarchy is shown in figure 2.8.

The functionality and implementation of all the VuDP modules will be described in more detail in later chapters.

2.3 Distributed Programming with VuDP

Distributed programming in the VuSystem involves writing applications that use remote sources, remote filters, remote sinks, and/or the VsNet modules and the VsTcp modules in order to distribute the modules of a VuSystem application over multiple execution environments on multiple hosts. The mechanisms used by a given application are dependent on the specific needs of the application.

2.3.1 Remote Sources

Traditional VuSystem Sources

Before the VuDP extensions were added, VuSystem applications could only access media sources that were directly available to the local host on which an application was running. For example, a VsPuzzle application running on host X could only access video sources directly available to host X.

This constraint was eased a bit by the use of NFS and the VuNet. The VuNet is the high-speed network companion to the VuSystem with the property that all sources on the VuNet are directly accessible by any host attached to the VuNet. Hence, any source on the VuNet is available to any host on the VuNet. NFS can be used to access remote files, and so VuDP is not needed to access remote file sources. However, a general remote sourcing mechanism is still desirable so that applications can access non-file sources that are not accessible via the VuNet.

In the context of the VuSystem, a "source" is a media source, generally an audio or video source. A VuSystem application (such as VsPuzzle, described earlier) which uses a media source typically specifies the name of the source it wants to use on the command line. For example, video sources are specified by the "-videoSource" option, and audio sources are specified by the "-audioSource" option. (Default audio and video source values for each host are used if no source name is specified.)

The syntax for specifying a source name is a colon (":") followed by the name of the source. For example, to start a VsPuzzle application and specify that it use the source named "vidboard1", the command "vspuzzle -videoSource :vidboard1" would be used. (Of course, without the VuDP extensions, vidboard1 must be available to the host running the VsPuzzle application.)

Naming Remote Sources

Using VuDP, the VuSystem sources (generally video and audio sources) are able to reach out across the network to access any source available to any host on the network. To handle this, the naming of VuSystem sources includes an (optional) host name before the colon preceding the source name. If no host name is specified for a source, it defaults to the local host. Hence, it is not necessary to modify existing VuSystem applications to use this source naming syntax. As a side benefit, the naming syntax allows for sources on different hosts to have the same local names.

For example, say that an application wants to use video board "vfc0" on host Y as its video source. The video board is not directly accessible, but (using the remote sourcing mechanism) the application can reach it by specifying "-videoSource Y:vfc0" on the command line to say that it wants to use the source "vfc0" which is accessible from host Y. (An application on host Y that wants to access the video source can specify either "-videoSource Y:vfc0" or "-videoSource :vfc0".)

Remote Sources and Dynamic Reconfiguration

Remote sourcing is more complicated than it first appears because of the reconfiguration capabilities of VuSystem applications, which (via the VuSystem control panel mechanism) allow the user of an application to dynamically reconfigure video and audio sources. Further, the VuSystem interactive programming environment allows users to examine, create, modify, and destroy any and all modules within an application. Remote sources must appear to be local for both the control panel and the interactive programming mechanisms.

For the interactive programming environment, stubs are used to relay commands to the remote sources over the network. This works very well, and, as far as interactive programming is concerned, remote sources appear to be local. Keeping the control panel interface consistent is more difficult, since the traditional VuSystem control panel implementation works by passing each primitive source an X-Windows widget in which to draw its control panel functions. Because X-Windows widgets can not be easily passed between address spaces, remote control panels must necessarily involve either completely redesigning and reimplementing the way control panels work or modifying the appearance of control panels for remote sources.

It was decided to slightly modify the appearance of the control panels for remote sources. Instead of having all controls for all modules appear in one control panel, the controls for a remote source appear in a separate control panel. All other features of the control panel mechanism are unchanged; the control panel for the remote source can be used to change and reconfigure the source just as if it were a local source. The only difference is that the controls for the remote source appear in their own window.

The appearance of the control panels for remote sources is illustrated in figure 2.9.

2.3.2 Remote Sinks

Traditional VuSystem Sinks

There are generally three kinds of media sinks in the VuSystem: window sinks, file sinks, and special device sinks. For these first two kinds of sinks, no remote sinking mechanism is needed because the VuSystem is built on top of X-Windows and NFS. Assuming proper access control settings, X-Windows allows a window to be placed on any display, so remote window sinking capability existed prior to VuDP. Similarly, NFS allows any host to access and manipulate files exported by any other host.

On the other hand, non-file and non-window sinks - such as the DEC J300 Sound and Motion Board, which can produce analog output from digital input - are accessible only from the hosts they are attached to. Hence, a general remote sinking mechanism is valuable for sinks other than files or windows.

Naming Remote Sinks

VuSystem applications specify the video sink to use by the "-videoSink" command line option, followed by the name of the video sink prefixed by a colon. For example, to specify a window video sink, an application would have "-videoSink :window" in its command line. If no video sink is specified, the default is typically a window sink of the display for that application.[3]

Using VuDP, applications can access any media sink available to any host in the network by adding an (optional) host name preceding the colon in the name of the sink. For example, if an application wants to use the J300 sink on host Y, it specifies "-videoSink Y:j300" on the command line.

Remote Sinks and Dynamic Reconfiguration

As for remote sources, remote sinks have local stubs which reroute commands from the interactive programming environment to the location of the actual sink. This works well, and (from the user's point of view) local and remote sinks are indistinguishable in the interactive programming environment.

Unfortunately, the same problems exist for both the remote sink and the remote source control panel mechanisms. The same design decision was made for remote sinks as for remote sources - the way control panels look for remote sinks was changed slightly to avoid modifying the VuSystem infrastructure and substantially complicating VuDP.

A remote sink has its own control panel in a window separate from the main control panel. All other features and functionality are the same; the control panel for the remote sink can be used to modify and dynamically reconfigure the remote sink as if it were a local sink.

2.3.3 Remote Filters

Traditional VuSystem Filters

A filter is any module with one input port and one output port. Typically, filters perform some processing operation on the media that flow through them. Examples of VuSystem filters are the VsPuzzle module (which scrambles a video stream), the VsContrast module (which allows for video contrast adjustment in software), and the VsEdge module (which performs edge enhancement on a video stream.)

[3] Note that since the VuSystem is built on X-Windows, the display that an application uses may not be the display physically attached to the host on which it is running. The display may be different if either the application is run in an X terminal running on a remote machine or if the application is run with a "-display" command line option.

The VsRemoteFilter module

The VsRemoteFilter module allows an application to place a filtering module on any host in the network. The VsRemoteFilter module can be used to create and place a filter on any machine, as follows. At creation, the VsRemoteFilter module takes two arguments, a host and a filter type. The filter type must be supplied, but if no host is supplied then the local host is taken as the default. The VsRemoteFilter then creates the appropriate type of filter module on the specified host and sets up in-band and out-of-band connections so that media automatically flows from the local application to the remote site, through the filter, and then back again to the local application.

VsRemoteFilter modules can be manipulated as if they were modules of the underlying filter type. For example, a VsRemoteFilter module that is used to instantiate a remote VsPuzzle will have all the VsPuzzle commands and options and can be manipulated and reconfigured as if it were a VsPuzzle module. Any VsPuzzle method invoked on the VsRemoteFilter module will be automatically passed on to the underlying VsPuzzle module.

Using the VsRemoteFilter module, applications can place filters in media streams and have the filtering operations performed on any host in the network. Because filtering operations are often compute-intensive, placing a filter on a remote machine can help to move work to a lightly loaded or faster machine.[4]

Remote Filters and Dynamic Reconfiguration

The VuSystem control panel and runtime interactive programming mechanisms can be used to dynamically reconfigure remote media filters.

Remote filters have local stubs which reroute commands from the interactive programming environment to the remote host where the underlying filter module exists. This allows remote filters to be reconfigured as if they existed in the local address space. Hence, to the user, a remote VsPuzzle filter can be manipulated exactly as if it were a local VsPuzzle filter.

For sources and sinks, it was decided that it is better to slightly modify the appearance of the control panel for remote filters than to rewrite a substantial part of the VuSystem infrastructure. Hence, for filters on remote machines, the controls for the remote filter appear in a separate window.[5] Except for the separate appearance of filter controls for remote filters, the control panel works the same for remote filters as it does for local filters.

---

[4] This is especially useful in situations where there are a few extremely fast computers attached to a network that all users can take advantage of to perform remote filtering while still running applications on their (slower) local machines. Of course, placing a filter on a remote machine increases communication costs because the media must be shipped over the network to the remote machine, filtered, and then sent back.

[5] However, if a VsRemoteFilter module is used to create a filter on the local machine, the controls will be integrated with the rest of the local control panel.

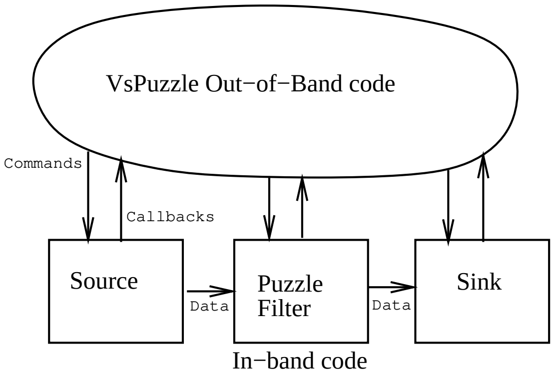

Figure 2.4: In-Band/Out-of-Band Partitioning in VsPuzzle

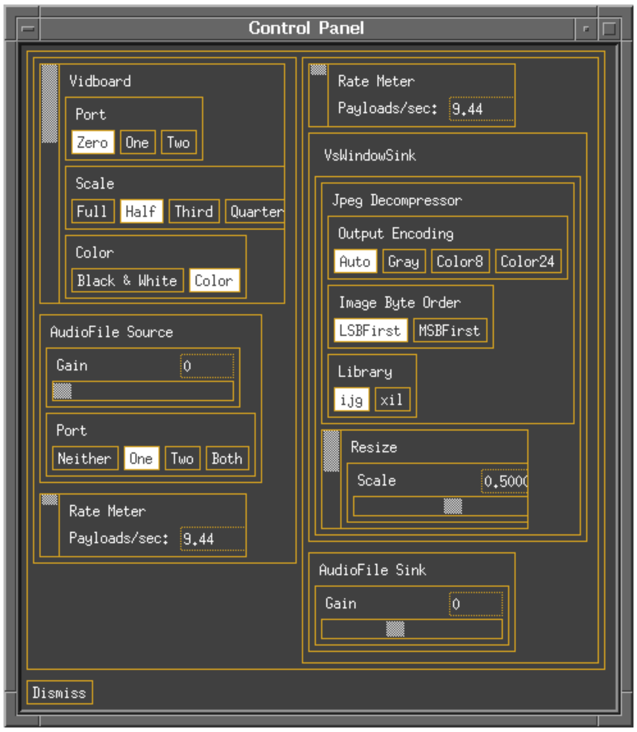

Figure 2.5: A VuSystem Control Panel

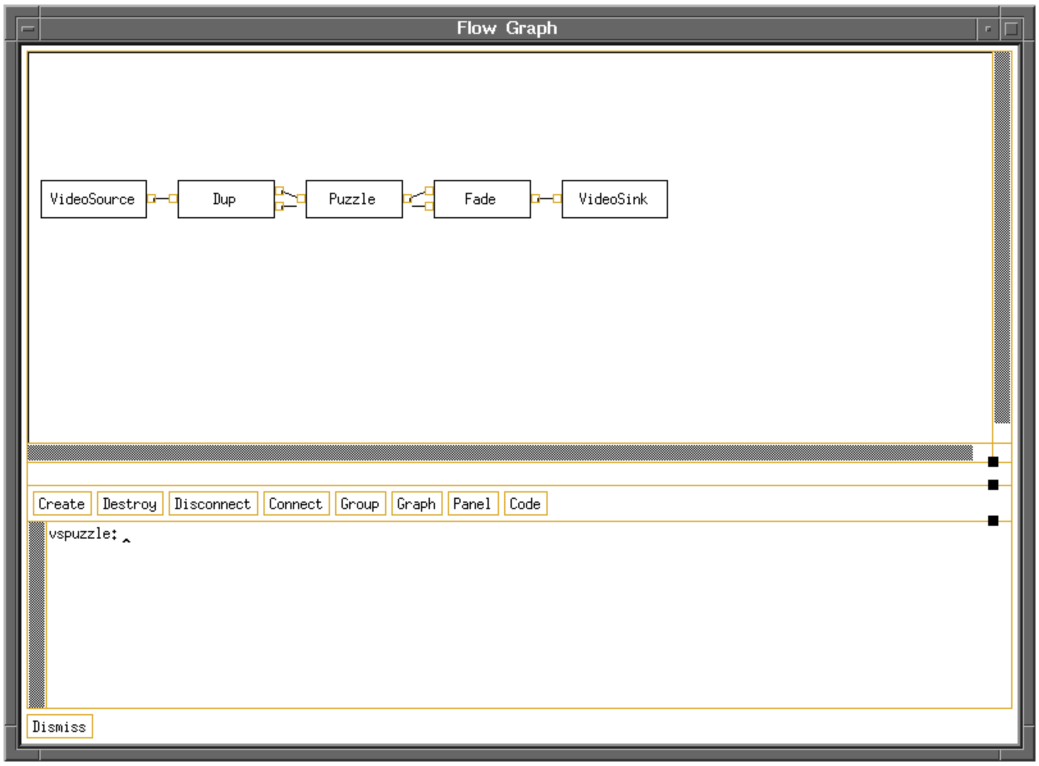

Figure 2.6: The Visual Programming Environment

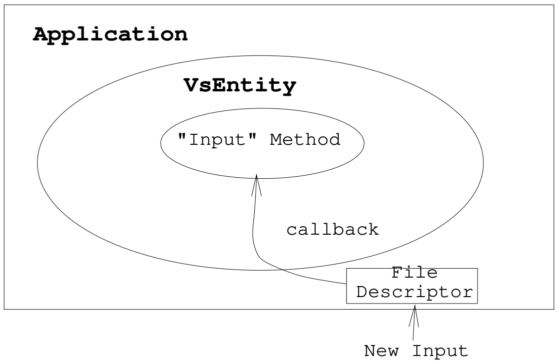

Figure 2.7: VsEntity Input Callback

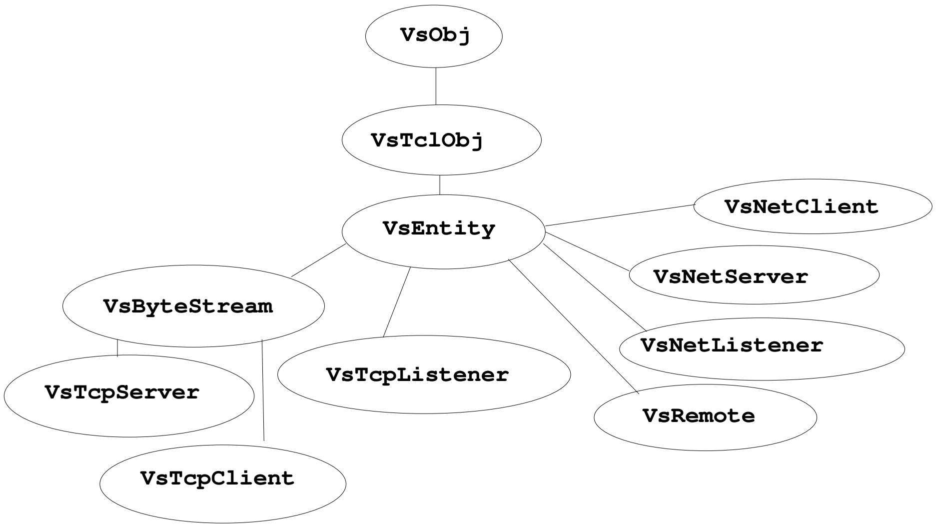

Figure 2.8: VuDP Hierarchy

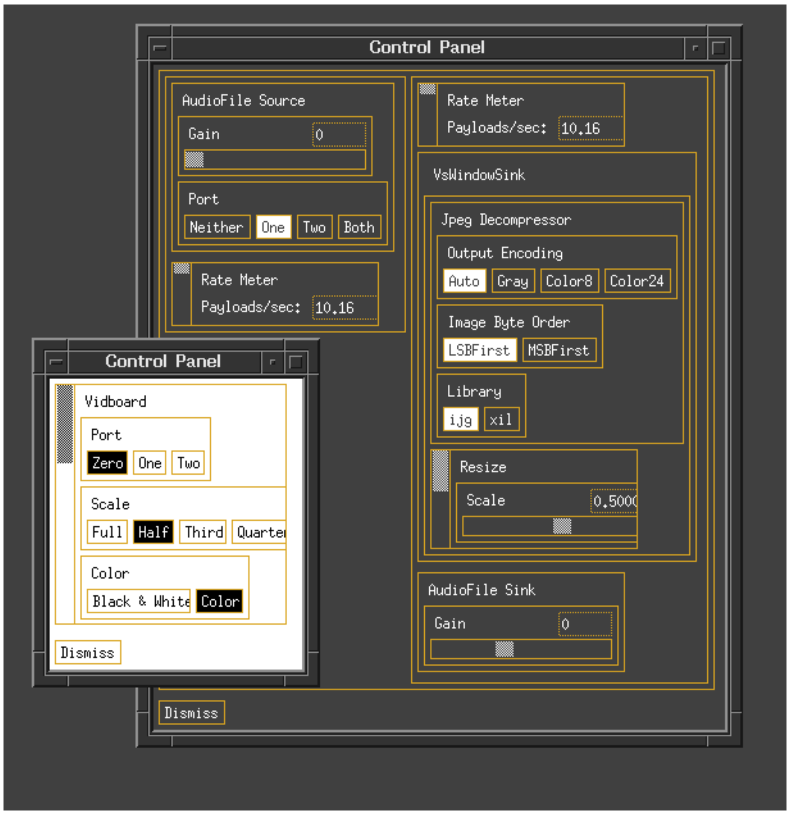

Figure 2.9: A Remote Source Control Panel

Chapter 3

The In-Band Partition

The VuSystem separates applications and their component modules into in-band and out-of-band code. This chapter discusses the in-band partition of VuDP that deals with the flow of media information between modules in a VuDP application.

VuSystem and VuDP applications have access to two underlying networks: a standard ten megabit per second Ethernet and the VuNet, a gigabit ATM network. All VuDP remote in-band communication facilities can be configured to work over either the Ethernet or the VuNet. Because the VuNet runs much faster than the Ethernet, applications generally prefer to use the VuNet for remote in-band communication.

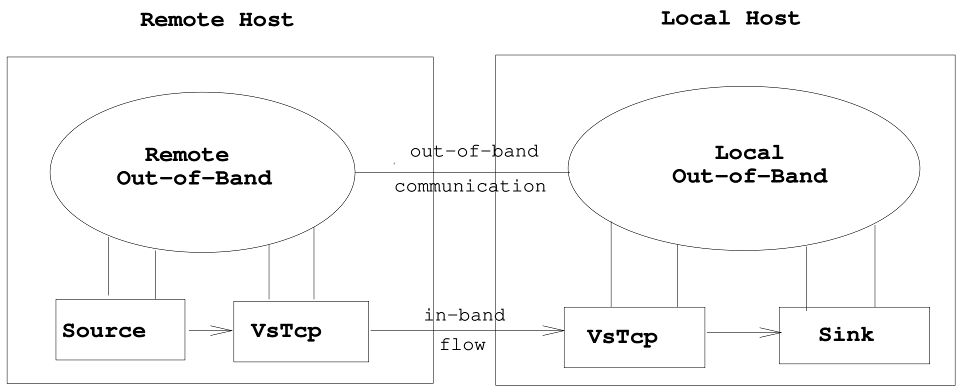

Figure 3.1: Remote In-Band Connections

3.1 The VsTcp Modules

VuDP applications can use the VsTcp modules to use the network to send in-band media streams to modules in remote execution environments on remote hosts. The VsTcp family of modules consists of the VsTcpClient, the VsTcpListener, and VsTcpServer modules. They are used to set up media flows between applications running in different address spaces which are typically (but not necessarily) on different machines.

3.1.1 Setting Up In-Band Connections

The VsTcpClient module is used to initiate the creation of an in-band communication channel between remote interpreters. Once created, a VsTcpClient module must be configured with the host name and port that it will connect to. When the VsTcpClient module is started, it attempts to make a connection to the specified host and port. It is the responsibility of the applications programmer to select appropriate port numbers for in-band connections.

The purpose of the VsTcpListener module is to wait for VsTcpClient modules trying to connect to the port at which it is listening. After creation, a VsTcpListener must be configured with the port it will listen at, and once started it will listen at that port and wait for remote VsTcpClient modules attempting to connect to it. When a VsTcpClient module tries to connect to a port where a VsTcpListener is waiting, a connection is made, and the VsTcpListener creates and starts a VsTcpServer module to handle the connection.

When the connection is established and the VsTcpListener creates a VsTcpServer to handle it, the VsTcpListener executes a callback to inform the application script that a connection has been made and to tell it the name of the new server it has created. Hence, any application which uses a VsTcpListener must set up a callback, or it will know neither when a connection has been established nor the name of the VsTcpServer which is handling the connection.

The VsTcpClient and the VsTcpServer modules are used to send and receive data across an in-band connection. Communications over an in-band connection are completely symmetrical, so the VsTcpServer has the same ability to send and receive data as the VsTcpClient. The difference between the VsTcpServer and the VsTcpClient modules stems from the setup of in-band communication channels - the VsTcpClient initiates a connection, while the VsTcpServer is created in response to a connection request.

Once a connection is established, the VsTcpClient module can be stopped and then reconfigured and restarted to initiate another connection. On the other hand, VsTcpServer modules are useful only for the duration of the connection they are created for.

Once started, VsTcpListener modules persist and listen for connections and may handle an arbitrary number of connections and start an arbitrary number of

VsTcpServers. Once stopped, VsTcpListener modules can be reconfigured and/or restarted to begin listening again. Like VsTcpClient modules, VsTcpListener modules can be reused.

3.1.2 Handling In-Band Connections

The VsTcp modules are meant to handle in-band media flows, and so the VsTcp modules can be viewed as special VuSystem filters. A typical VuSystem filter module has one input port and one output port, and when the input and output ports are connected and the filter module is started the media flowing into the input port is filtered and sent out the output port. VsTcp modules are filters whose processing operation is to send the media over the network. The media that flows into the input port of a VsTcpClient module is sent over the connection and comes out of the output port of the VsTcpServer module on the other end. Similarly, the media that flows into the input port of a VsTcpServer module is sent over the network and flows out of the output port of the VsTcpClient module it is connected to. There are no "send" or "receive" commands for VsTcp modules. Instead, the media flow between the modules is governed by the VuSystem Module Data Protocol.[14][1]

As the names suggest, VsTcp modules use the TCP communication protocol, and so neither the modules nor the applications using the VsTcp modules need to worry about packet loss, reordering, or other reliability problems of the underlying network.

3.1.3 Example of an In-Band Connection

Figures 3.2 and 3.3 illustrate the creation and use of an in-band connection between two interpreters: interpreter A, running on host X, and interpreter B, running on host Y. (As for the out-of-band example, these interpreters may be two separate applications or two interpreters that exist within a single distributed application.) First, the interpreters are running in separate name spaces. Interpreter B then creates a VsTcpListener, configures it to listen at port 9000 on host Y, and starts the module. Next, interpreter A creates a VsTcpClient module and configures it to connect to port 9000 on host Y.

Interpreter A then starts the client module, which establishes a connection to the listener module in interpreter B. When the connection is made, the VsTcpListener module starts a VsTcpServer to handle the connection, as shown in figure 3.2.

Finally, the input and output ports of the VsTcpClient and VsTcpServer modules are connected appropriately, and the automatic flow of media between the interpreters begins as shown in figure 3.3.

The connection persists until either the user of interpreter A stops or destroys the VsTcpClient (perhaps by exiting), or interpreter B exits or otherwise destroys the

[1]This will be discussed in further detail later in this chapter.

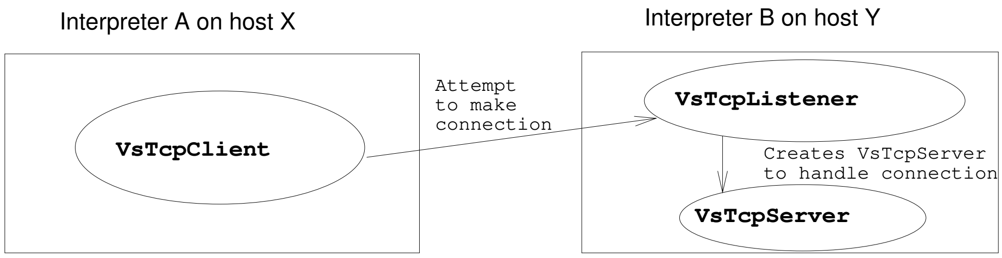

Figure 3.2: In-Band Connection Initiated

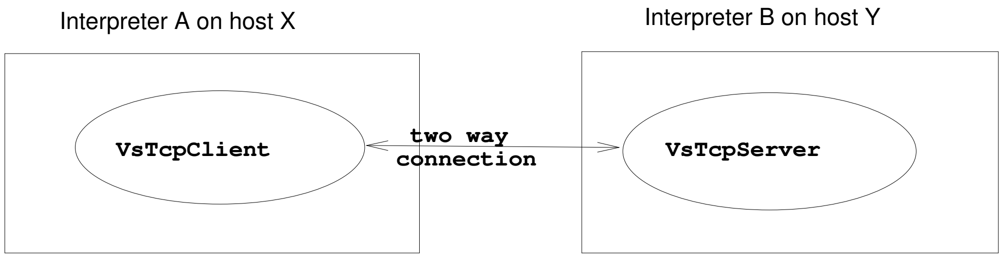

Figure 3.3: In-Band Connection Established

VsTcpServer. If a connection is terminated by the destruction of the VsTcpServer module on interpreter B's side, an error message is relayed back to interpreter A.

3.2 Implementation

Most of the functionality of the VsTcp module was implemented prior to VuDP as part of the traditional VuSystem.[2]

The one major change VuDP made to the VsTcp modules was to add the ability to specify either the VuNet or the ethernet as the underlying network. A new option command, the "net" command, allows an application to specify either "ethernet" or "vunet" as the network that the VsTcp modules use for communication.

[2]See [14] for more details.

3.3 In-Band Communication Protocols

3.3.1 The VuSystem Module Data Protocol

The VuSystem module data protocol [14] is a closed-loop protocol that governs the transfer of in-band data between modules in a traditional VuSystem application. (That is, between modules executed by the same thread.) The protocol provides for tightly-coupled flow control to match the ability of the upstream module to send "payloads" (the unit of media data transfer) to the ability of the downstream module to receive "payloads".[3]
Whenever an upstream module has data to send, it calls the "send" procedure on its output port, which attempts to send a payload over the connection by calling the "receive" procedure of its downstream module. If the downstream module accepts the payload, the send is successful, and the upstream module is free to continue to perform more sends until it is rejected. But, if the downstream module rejects a send, it is not ready to receive new data, and so the payload is not sent and the send is unsuccessful. By rejecting a send, the downstream module commits itself to informing the upstream module when it is ready to receive a new payload. When the downstream module is finally ready for the new payload, it is obliged to call the "idle" procedure of its input procedure, which in turn calls the "idle" procedure of the upstream module, which causes the upstream module to try "send" again if it has data to send. If the upstream module has no data to send, then it "starves" the downstream module by not sending any data in response to the "idle".

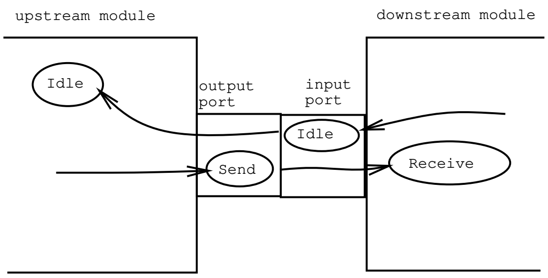

Figure 3.4: The VuSystem Module Data Protocol

Note that an upstream module may continue to try to send a payload after a send has been rejected and before the downstream module has called idle. However, such sends are likely to be rejected. Similarly, the downstream module may continue to call idle when it has not received any new data, though if the upstream module still has no data to send the extra idle calls will have no effect.

[3] A payload typically corresponds to a single video frame or segment of audio data.

The VuSystem module data protocol provides tightly coupled closed-loop flow control. Only one payload may be in transit at any time, and flow control is done on a payload-by-payload basis.

3.3.2 Extending the Module Data Protocol

The VuSystem module data protocol works well in the context of the traditional VuSystem, where all modules exist within the same address space. The latency of inter-module communication is virtually nil, and (generally) passing a payload involves no more than copying a pointer, so payloads can be passed quickly between modules.

The situation becomes more complicated with VuDP because the characteristics of inter-machine communication over a network are much different than simple inter-module communication within an address space. When sending data across a network both end-to-end flow control (matching the rates of the upstream and downstream modules) and congestion control (matching the rate at which the underlying network can deliver payloads) are important.

Because of the differences between inter-module communication within a traditional VuSystem application and inter-module communication over a network within a VuDP application, it is not reasonable to directly extend the VuSystem module data protocol to work over a network. First, the latency of transferring payloads over a network causes lock step flow control to have terrible performance. With the traditional VuSystem module data protocol, the window size is one, meaning that at most one payload may be in transit at any time. Once a payload is sent, the upstream module must wait for the network to send and deliver the payload to the appropriate machine, for the machine to deliver the payload to the application, for the application to accept or reject the packet, for the network to deliver the response back, and for the host machine to deliver the response back to the application. Each of these components contributes to latency, as summarized in figure 3.5. To make things worse, some of these components of latency are highly variable, such as network delay. Taking all these considerations into account, the round-trip time of intermodule communication is far too high for the traditional VuSystem module data protocol to provide decent performance for passing payloads between two modules in separate address spaces on separate machines connected by a network.

A second issue is that the traditional VuSystem module data protocol is implicitly lossless. Because all data transfers occur within an address space, there is no chance that payloads will be lost while traveling between modules, and the protocol depends on this implicit reliability. As shown in figures 3.6 and 3.7, loss of a payload in the traditional VuSystem can lead to deadlock. If the upstream module has starved the downstream module and then the next payload sent is lost, deadlock results.

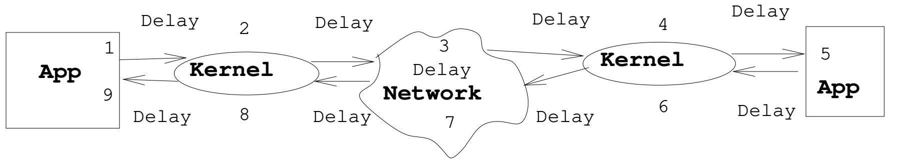

Figure 3.5: Components of Latency

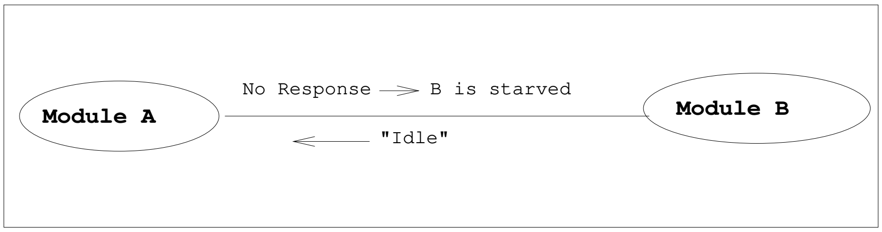

Figure 3.6: B Calls "Idle" and is Starved

3.3.3 The VuDP Module Data Protocol

TCP

VuDP solves the problems with the traditional VuSystem Module Data Protocol by using a TCP connection between upstream and downstream modules in different execution environments. TCP provides for efficient, reliable transport that provides both flow control and congestion control.
TCP addresses the latency problem by buffering data to be sent and using a sliding window to send data in a pipelined manner so that it is possible for many packets to be in transit at any given time, better utilizing the bandwidth of the network. TCP also uses its sliding window and a timer to handle both end-to-end flow control and network congestion control.[8] Finally, TCP provides reliable stream delivery by retransmitting lost and delayed packets and reordering packets at the receiving end so that the receiving application is presented with the data in the same order that the sending application sent it. The details of TCP are well documented [22] and will not be described here.

The VsTcp Modules

To see how the TCP connection works with the VuSystem Module Data protocol, consider a upstream video source trying to send a payload to the downstream

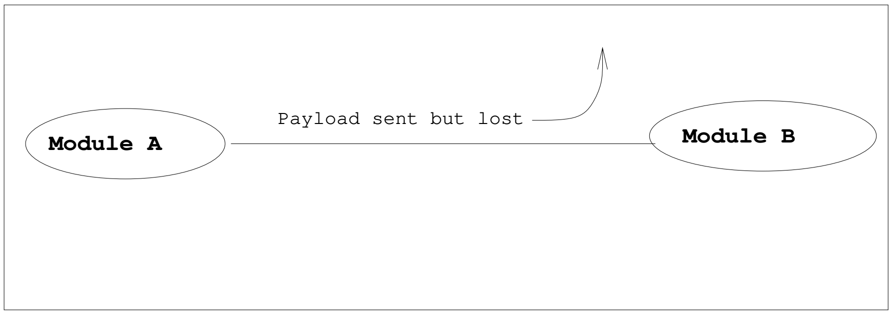

Figure 3.7: A Sends a Payload which is Lost

VsTcpClient module. The VsTcpClient will accept the data, buffer it for transmission, and reply to the video source that the payload was accepted.[4] When the VsTcpClient accepts more payloads than it can buffer, it is necessary to look more closely at the socket interface presented by the BSD TCP implementation [8]. Once a connection is established, the VsTcp modules see the connection as a file descriptor. If a VsTcp module attempts to write more data than can be stored in the connection buffers, the process is blocked until enough buffers are available to accept the new data. Hence, the VsTcpClient module in the example will never reject a payload. But, if it accepts a payload that the underlying connection does not have enough buffer space to store, the process will be blocked until enough buffer space is available.

Once payloads have been written to the socket by the VsTcpClient module, they are sent via the underlying TCP service, and appear as new readable data in the connection socket for the VsTcpServer module in environment B. TCP is a reliable stream protocol, so the data will always be delivered to the application (in this case a VsTcpServer module) reliably and in same order it was sent. Then, when it has constructed a new payload by reading data from the socket, the VsTcpServer module will attempt to send the payload to the video sink module in environment B using the traditional VuSystem module data protocol. The underlying TCP service makes the connection socket appear to be a file descriptor, so the VsTcp modules can read payloads from and write payloads to a connection in exactly the same way that payloads are written to and read from disk files.

At a high level, using the VsTcp modules allows VuDP to appear to seamlessly extend the VuSystem module data protocol for use across execution environments. Modules that are either upstream or downstream from VsTcp modules interact with them via the traditional VuSystem module data protocol. To all modules

[4] Note that this is one way in which the VsTcp modules differ from traditional modules, since the VuSystem module data protocol does not call for any buffering.

either upstream or downstream from the TCP connection it appears that all the modules in the media flow communicate using the traditional VuSystem Module Data Protocol. All the details of using TCP are hidden within the VsTcp modules. The apparent seamless extension of the VuSystem module data protocol is a powerful feature of the VuDP module data protocol.

3.3.4 Alternatives: Lossy Transport Protocols

Both the traditional VuSystem and the VuDP module data protocols are reliable (lossless) protocols. However, for VuDP this reliability incurs an overhead. The network may lose or delay packets and cause them to be retransmitted, and so work needs to be done to make sure that all payloads are delivered reliably and reordered appropriately. This work both complicates the implementation and hurts the performance since when packets are lost or delayed new data can not be sent beyond a certain point until all previous data has been retransmitted and received successfully. These delays are inherent to a lossless protocol but may be costly especially since media data such as video streams require very high bandwidth and are often very sensitive to jitter introduced by variable delays between frames.

However, it is often not necessary to use a lossless protocol for media data, especially video. The nature of video is such that it is sometimes acceptable to drop frames, and that it may be better to simply drop a frame than to display frames out of order or to delay the display of new frames because previous frames (or portions of previous frames) have been lost or delayed.[5]

In order to put these concepts into a more concrete framework, take lossless/lossy and lockstep/window-based flow control as two distinguishing characteristics for transmission protocols. This provides for four different types of protocols, as shown in figure 3.8.

The traditional VuSystem module data protocol is an example of lossless protocol with lockstep flow control (i.e. window size equals one.) This is reasonable within an application, as payloads are never lost (so reliability costs nothing), and communication latency is low enough that there is no performance penalty for using lock step flow control.

TCP (and hence the VuDP module data protocol) is an example of a lossless protocol with window-based flow and congestion control. The sliding window provides flow control, network congestion control, and reliability (via retransmission of lost and delayed packets) while maintaining good performance through pipelining. The reliability of the protocol is not costless, though.

The Vidboard protocol [2] is an example of a lossy protocol with windowless closed-loop flow control. The Vidboard is a hardware device that connects to a primitive video source (typically a camera, a VCR, or a television feed) and

[5] Using a simple protocol such as UDP is not acceptable, since the protocol must still provide adequate flow control and network congestion control.

Lock-Step Flow Control **Lossless** **ex: VuSystem MDP**	**Window Flow Control** **Lossless** **ex: TCP, VuDP MDP**
Lock-Step Flow Control **Lossy** **ex: Vidboard**	**Window Flow Control** **Lossy**

Figure 3.8: Types of Protocols

digitizes the incoming video into frames. The input of the Vidboard is a constant bit-rate video stream, but the output of the Vidboard must obey the VuSystem module data protocol, meaning that it can only pass payloads downstream when the downstream module is ready for them. To handle these conflicting demands, the Vidboard provides minimal buffering and saves only the most recently digitized frames in its memory. A frame which is created but never passed downstream because a new frame was written over it before a request for new data was received is simply lost. The Vidboard thus discards video frames as needed and so operates according to a lossy protocol in order to match the open-loop constant bit rate of its input to the closed-loop dynamically varying capacity of its output.

The VsLossyDup Module

In the traditional VuSystem, the VsDup module has one input port and two output ports and copies its input stream to produce two identical output streams. It abides by the VuSystem Module Data Protocol by waiting until both of its output ports have sent the payload downstream before it accepts a new payload. This forces both of its output video streams to operate at the rate of the slower of the two. Normally, this isn't a real problem, but in applications such as VsMultiCast[6] (where there may be many VsDup modules hooked together) the media flow grinds to a halt since the entire stream must travel at the rate of the slowest module. The above performance consideration motivates the development of a duplicator module that uses a different protocol for pacing its output streams. VsLossyDup is such a module. It appears to be the same as a VsDup module - one input port,

[6]See chapter five for more details on VsMultiCast.

two output ports, with the input stream copied and produced at both output ports - but uses a different algorithm for handling its output. The basic idea is to duplicate the Vidboard lossy lock-step protocol for handling payloads. The VsLossyDup module will never reject a payload just because it hasn't sent it downstream to both output ports. Instead, it always accepts new payloads sent to it by upstream modules. Further, it always supplies the downstream modules with the most recent payload it has received, while ensuring that it never sends the same payload to the same downstream module twice.
The rules that the VsLossyDup module follows for handling payloads can be summarized as follows:

- Always accept payloads sent from the upstream module, and attempt to send the new payload to both downstream modules when a new payload arrives.
- If a new payload is received before the old payload has been sent to both downstream modules, discard the old payload.
- When either downstream module calls "idle", check to see if the current payload has already been sent to that module. If not, send it.

3.3.5 A Lossy VuDP Module Data Protocol

One way to possibly improve performance of VuDP applications is to develop a lossy protocol that provides the pipelining and congestion and flow control of TCP. This would eliminate the overhead of providing reliable transport, hopefully improving throughput and reducing jitter. The gains from such a protocol are limited by a few factors, though.
Most of TCP's retransmit mechanisms are closely entwined with the flow and congestion control mechanisms. In order to provide good flow and congestion control it would still be necessary to acknowledge every message and keep a timer at the sender's end in order to determine the proper rate to send data. So, a lossy protocol would most likely be even more complex than TCP. Protocols such as TCP typically declare a packet "lost" after a certain timeout expires. TCP uses such a timer at the source, but a lossy protocol would require a timer at the receiver in order for it to determine when packets are lost. So, jitter would not be entirely eliminated, and some throughput would still be lost when packets were lost, since it would be necessary to wait for an appropriate timeout to expire before declaring a packet lost. Clearly, setting an appropriate timeout would be extremely important, as too long a timeout would cause unnecessary jitter and transmission delays, while too short a timeout would declare slightly delayed packets as being lost. Finally, it would be necessary to have some way of informing the application of lost packets. This is because video frames are likely to be split into several packets for transmission over the network, and if any packet within a video frame is lost then the entire frame must be discarded. Thus, the application receiving the

data needs to have an additional mechanism to know how to handle video frames (or audio buffers or whatever the underlying media is) that have portions missing. Despite these problems, a lossy transport protocol for media data has the potential to improve the performance of VuDP applications that must send in-band media flows over a network. The question of whether or not VuDP actually has any performance deficiencies that might be improved by using a lossy protocol will be addressed in chapter six.

Chapter 4

The Out-of-Band Partition

This chapter discusses the out-of-band partition of VuDP. The out-of-band code creates, configures, and manipulates the in-band modules that manipulate media data. The VuDP remote source, remote filter, and remote sink implementations consist entirely of out-of-band code.
Though VuDP applications may use either the Ethernet or the VuNet for out-of-band communication across the network, out-of-band communication is generally very low bandwidth and hence the throughput of the underlying network is not important.

4.1 The VsNet Modules

The VsNet modules (VsNetListener, VsNetClient, and VsNetServer) are used for setting up out-of-band message-passing communication across the network. The VsNet family consists of three modules: the VsNetClient, the VsNetListener, and the VsNetServer. These modules can set up out-of-band TCP communication channels between interpreters in separate address spaces. In other words, they are used for the exchange of Tcl data (such as short command strings) and configuration and control information between remote modules. The VsNet modules are not designed for sending media over the network.

4.1.1 Setting up Out-of-Band Connections

The use of the VsNetClient, VsNetListener, and VsNetServer modules for setting up connections almost exactly parallels that for the VsTcp modules.
In order to set up a connection, a VsNetClient must be configured with a port and host to connect to, and an underlying network to use. When the client module is started it attempts to make a connection to a VsNetListener module at the specified host and port over the specified network. On the other end, a VsNetListener module must be configured with a port number to listen at, and

when started will listen at that port until a remote VsNetClient attempts to make a connection to it. When a connection is established, the VsNetListener starts a VsNetServer module to handle the connection and informs its controlling application via a callback.

4.1.2 Handling Out-of-Band Connections

Once an out-of-band connection is established, VsNet commands can be used to send and receive data. The commands for sending and receiving data - “send” and “call” - are exactly the same for the VsNetClient and VsNetServer modules, as out-of-band connections are two-way and completely symmetrical.

The “send” and “call” commands can be used to push data across a connection. The “send” command is non-blocking and sends data asynchronously; the sending application continues to run after the “send” command and the data is sent in the background. By contrast, the “call” command sends data by blocking the calling application until either a return value is received or a time-out expires. (If the time-out expires before any return value is received, an error is reported.)

The “send” and “call” commands take as their argument the data to be sent. The data must be a string; this is both convenient and powerful since strings are the only data type in Tcl. By manipulating the way the data to be sent is surrounded by double quotes and/or curly braces, variable substitution can be forced to be done either locally or remotely, as desired. As a general rule, substitution for any variable name that is surrounded by curly braces is performed in the remote environment, and all other variables are evaluated in the local environment. See [17] for more details.

To receive data from an out-of-band connection, an application must set up a callback routine that will be automatically called when new data is received. When new data arrives, the callback routine will be called with the argument “-command” whose value contains the data received and the “-call” argument which indicates whether or not the application that sent the data is blocking and waiting for a return value (i.e. whether the data was sent with a “send” or “call” command.) If a return value is expected, it is the responsibility of the receiving application to make sure that a return value is sent.

Applications can use the VsNet family of modules to freely mix synchronous and/or asynchronous communication as desired.

The VsNet family of modules uses the TCP protocol for the underlying communication channel. Hence, the TCP protocol takes care of delivering the messages in order and resending potentially lost packets. This allows applications using the VsNet modules to focus on sending and receiving data without having to worry about unreliability in the underlying network.

4.1.3 Example of an Out-of-Band Connection

Suppose that VuSystem interpreter A (running on host X) wants to set up an out-of-band communication channel with VuSystem interpreter B, running on host Y. (These interpreters may be either two separate VuSystem applications or two interpreters within a distributed application.) Initially, the interpreters are separate. Interpreter B must first create and start a VsNetListener module which listens at a specific port - say, port 8000 - on host Y. Then, interpreter A must create a VsNetClient module, and configure it to connect to port 8000 on host Y. When interpreters A's VsNetClient module is started, it attempts to connect to port 8000 on host Y. Because interpreter B has a VsNetListener module listening to port 8000, a connection is established. The VsNetListener creates a VsNetServer module to handle the connection, and interpreter B is informed via a callback that a connection has been made. Interpreters A and B can then use their VsNetClient and VsNetServer modules to send and receive data over the connection.
The connection persists until either interpreter A stops or destroys its VsNetClient module, or interpreter B exits or otherwise destroys the VsNetServer module. If the connection is terminated by the destruction of the VsNetServer module on interpreter B's side, an error message is relayed back to interpreter A.

4.1.4 Other Features

Other VsNet module commands are the "host", "port", "flush", "net", "wait", "backlog", and "timeout" commands.
For the VsNetClient module, the "host" and "port"[1] commands allows it to select a host and port to connect to, and the "net" command allows it to select either the ethernet or the VuNet as the underlying network.
The VsNetClient and VsNetServer modules can use the "flush" command to block the application and immediately attempt to send all data enqueued by previous "send" commands.[2]
Three other useful VsNetListener commands are "port", "wait", and "backlog" command. "port"[3] sets the port number at which the module will listen."wait" blocks the controlling application until either a connection is made or the timeout expires. "backlog" is used to set the maximum number of pending requests that the VsNetListener will enqueue before the listener starts to turn away additional requests for connections.[4]

[1] The default port is the port that the VuDP daemon (described in chapter five) listens to.

[2] Because all VuSystem applications are single-threaded, the actual sending of data sent with "send" is done in the background. Hence, there is no way for an application to know if the data sent with a "send" has been sent yet. After a "flush" has been executed, the application is guaranteed that there has been an attempt to actually send all data sent by previous "send" commands.

[3] As for the VsNetClient module, the default port is the port that the VuDP daemon listens to.

[4] The default value for the backlog is ten.

All VsNet modules have a "timeout" command which can be used to set the timeout value. For the VsNetClient and VsNetServer modules, this controls the duration that the module will wait for a response to a "call" before signaling an error. For the VsNetListener module, the timeout controls the duration it will wait for a connection after a "wait" is issued before returning an error.[5]

4.2 Implementation

4.2.1 The VsNetListener Module

The "port", "wait", "listen", "backlog", and "timeout" commands are implemented as Tcl commands which are handled by procedures that are declared as friends of the VsNetClient class and included with the rest of the code for the VsNetClient implementation.
When the listener is started, it opens a socket on the desired port, sets the appropriate socket options, binds to the socket, and executes a "listen" system call to sit and listen to the socket. Because the socket is associated with a file descriptor, the VsEntity input callback mechanism can be used, and so the listener sets up its "Input" method to be called when there is input available at the socket. Wen the "Input" method is called, there is new input at the socket, most likely a remote VsNetClient attempting to make a connection. The listener then issues an "accept" system call. If the connection is established, the listener creates a VsNetServer module to handle the connection, and then notifies the Tcl script by evaluating its callback, passing the name of the newly created server module by adding a "-obj [name of server object]" to the end of the Tcl callback string. The listener then resumes listening to the socket.
The "stop" method works by stopping the input callback, so that any input coming to its socket is ignored until the listener is restarted. Changing the port of a started listener first stops the module, then changes the port, and finally restarts it listening at the new port.

4.2.2 The VsNetClient Module

The VsNetClient commands "send", "call", "flush", "port", and "host", and "timeout" are all implemented as Tcl commands that are linked to procedures which are friends of the VsNetClient class. There is no default host; a client must be configured with a hostname before it can initiate a connection.
When a VsNetClient module is started, it opens a socket, sets the socket options appropriately, and uses the "connect" system call to try to connect to the desired machine and port. (The socket is set up as non-blocking so that the timeout can

[5] The default value for the timeouts is 25 seconds.

be used.) If the connection is established, the client module also sets up its "Input" method to be called whenever new input is available on the socket. Stopping a VsNetClient stops the input callbacks and closes the socket. Changing the host or port of a VsNetClient causes the module to be stopped, reconfigured, and then restarted with the new values.
Figures 4.1 and 4.2 illustrate using a VsNetClient and a VsNetListener to set up an out-of-band communication channel. First, the client and server exist in separate address space in applications A and B on hosts X and Y, respectively. Then, the listener starts listening at port N, and the client is configured to attach to port N on host Y. When the client is started, it connects to the listener, which creates a VsNetServer module to handle the connection. After the connection is made the listener resumes listening to the same port and can be reused to instantiate other connections.

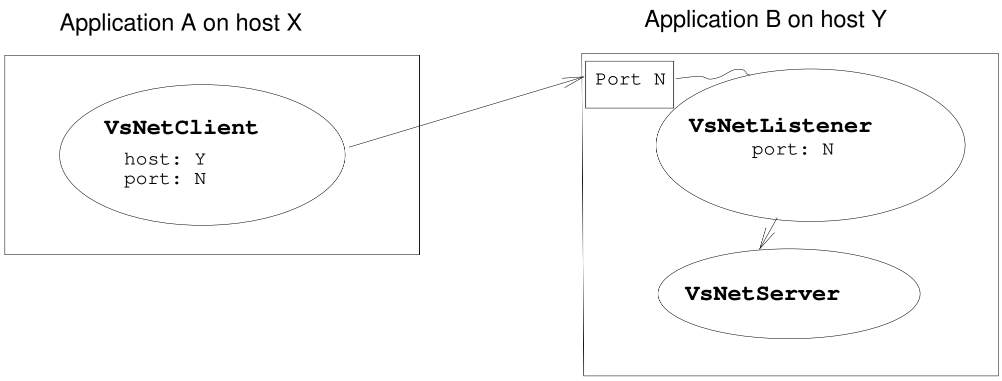

Figure 4.1: Connection Initiated

Asynchronous Send

Applications can use the VsNetClient "send" command to send data asynchronously. When a "send" is performed, the client module first computes the length of the message. It then prefixes the message with a five-byte header: the first four bytes represent the length of the message (in bytes), and the last byte signals that this message has been sent with a "send" command (as opposed to a "call" command.) The message is then put into a dynamically expanding circular queue maintained within the client module. Then, the VsEntity output callback mechanism is used to signal that output to the socket is desired, so that (at some indeterminate point in the future) the VsNetClient's "Output" method will be called to actually send the data over the socket. Hence, when a "send" command returns all that is known is that the message has been queued to be sent. The

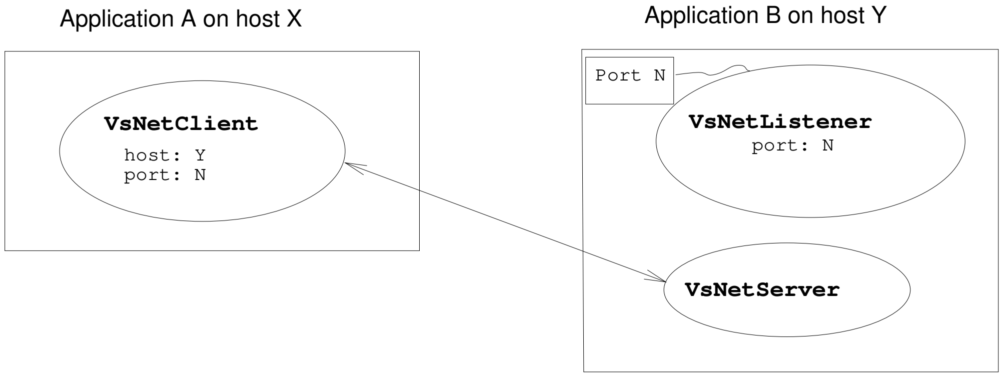

Figure 4.2: Connection Established

message will not actually be sent until the "Output" method is called in the background.

The circular queue in which unsent messages are maintained expands as needed. Because the queue doubles its size whenever it is about to be filled up, it is not possible to overflow it. The queue serves two purposes. First, because the VsEntity output mechanism executes the "Output" method in the background, it may be that multiple "send" commands are issued by an application before any of the data is actually sent over the socket, so the data must be stored somewhere. Second, transitory network problems may force the sending of messages to be delayed, so the queue can store the unsent messages until it is possible to send them.

When the "Output" method is called, it goes through the enqueued messages in order and attempts to send them over the connection using the "write" system call. Messages are deleted after they have been successfully sent. Each call to "Output" will continue to send enqueued messages, in order, until either the queue is empty or a message can not be sent.

When the "flush" command is called, it forces the VsNetClient module to send all of its enqueued messages by directly calling the "Output" method. Hence, when "flush" returns the application is guaranteed that it has tried to physically send the messages.

Asynchronous Receive

When a VsNetClient is started, it sets up its "Input" method to be called whenever new input is available at the socket. When the "Input" method is executed, it first uses the "read" system call to read five bytes from the socket to determine the length of the message and whether it was sent using "send" or "call". The rest of the message is then read. Then, the message is tagged with a sequence number

determined by simply adding one to the sequence number of the last message received. (The very first message received has a sequence number of one.) This is important, because Tcl gives no guarantees for the ordering of callbacks.
Lastly, the message is sent back to the application script using the VsEntity callback mechanism. The client module executes a callback with the arguments "-call", "-num", "-eof", and "-command". The "-call" argument is set to 1 if the message was sent by a "call", and set to 0 otherwise. The "-num" argument gives the sequence number of the message. The "-eof" is set to 1 if the connection has been closed, and set to 0 otherwise. (When a connection is closed, the "Input" routine will be called with zero bytes available. Hence, if the message is 0 bytes long, the connection has been closed, and "-eof" is set to 1. This allows the application to know when the connection has been closed.) Finally, the "-command" argument contains the body of the message that was received.
Note that it is the responsibility of the application script to make sure that a return value is sent for each "call" message received. When a message sent with "call" is received, the VsNetClient will set the "-call" argument in the callback to let the application know that the VsNetServer module on the other side of the connection is blocking and waiting for a reply. However, it is up to the application to send the response; if a response is not sent in a timely fashion, the timeout for the VsNetServer module will expire.
Figures 4.3, 4.4, and 4.5 illustrate asynchronous communication and the interaction of the "send" command, the message queue, the "Output" method, the input method, and the callback.

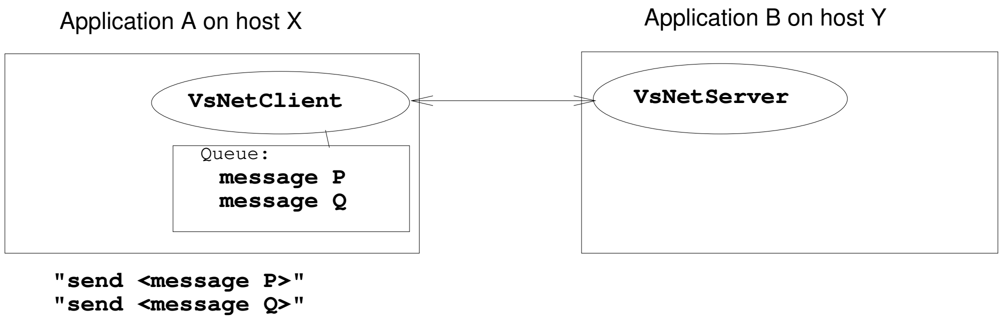

Figure 4.3: Messages Enqueued

Synchronous Communication

Synchronous communication can be performed by using the "call" command. When a "call" is executed, the module first temporarily stops all other

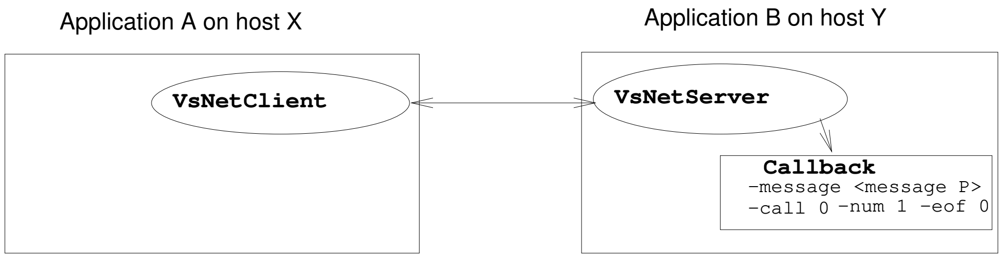

Figure 4.4: Message P Received, Callback Executed

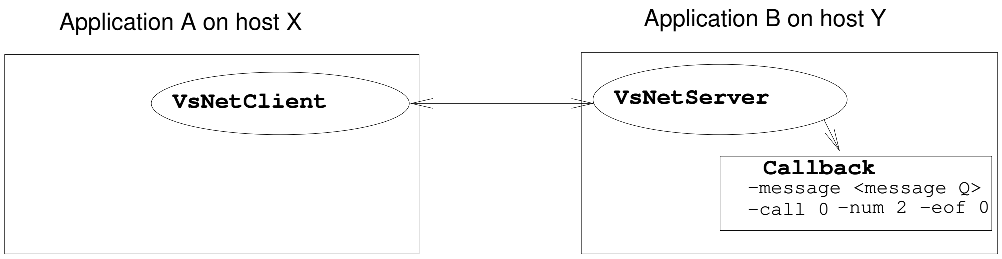

Figure 4.5: Message Q Received, Callback Executed

asynchronous input and output. Next, the module creates the five-byte header for the message; the first four bytes contain the length of the message, and the fifth byte signals that the message was sent with the "call" command. The "write" system call is then used to immediately send the message.
The module then blocks until either it receives new input over the socket or its timeout expires. If the timeout expires, the module stops blocking and generates an error message. If new input is received before the timeout expires, then it must be the response to the call, and is read from the socket with the "read" system command. Asynchronous input and output are then restarted, and the message received is returned to the application as the result of the "call" command.
By disabling asynchronous input and output, the VsNetClient module guarantees that the "Output" routine will not be invoked in response to the message sent in response to the "call". However, it is possible that some other asynchronous message sent to the VsNetClient will be mistaken as the response to the "call". In order to prevent this from happening, the application using the VsNetClient must make sure that it is not expecting any asynchronous input at the time it makes a

"call". This constraint has not been found to be a problem, since applications generally know when they are expecting asynchronous input.[6]
Figures 4.6, 4.7, and 4.8 illustrate an example of synchronous out-of-band communication.

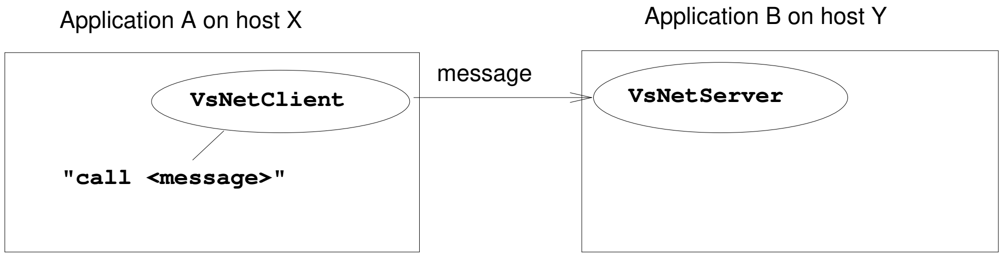

Figure 4.6: A Makes a Call to B and Blocks

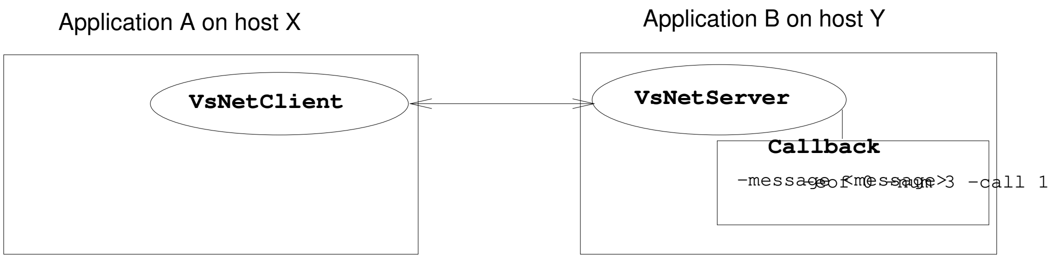

Figure 4.7: B Receives Message, Executes Callback

4.2.3 The VsNetServer

All out-of-band communication capabilities are two-way and entirely symmetrical. Hence, the VsNetServer and VsNetClient have the same communication commands ("send", "call", "flush") that work in exactly the same way.
Because the VsNetServer exists only to serve a single connection, it can not be configured for a specific host and port as a VsNetClient can. Further, when the connection is terminated the VsNetServer becomes useless. However, it is not automatically destroyed. Instead, the application using the VsNetServer is informed via a callback that the connection is broken, and the VsNetServer will

[6] Applications rarely (if ever) received unexpected asynchronous input since applications must explicitly set up constructs for receiving asynchronous input.

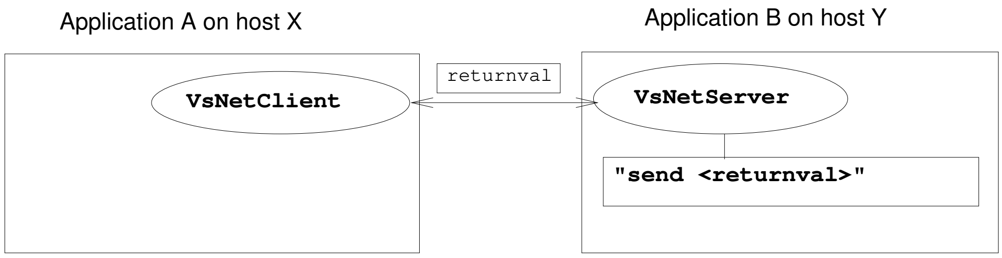

Figure 4.8: B Sends Back the Return Value to A

persist (as a useless module) until explicitly destroyed. As a general rule, applications should destroy VsNetServer modules immediately after being informed that the connection has been broken, unless there is a good reason for keeping it around (such as the module having surviving children.)

4.2.4 VsRemote

The VsRemote module exists for use only as a parent object for the VsNetClient and VsNetServer modules. VsRemote is a subclass of VsEntity, and provides most of the same functionality. The only difference is the way that "start" and "stop" are handled. In the traditional VuSystem, the "start" method for a VsEntity (and all subclasses of VsEntity) will automatically start the module and all of its children. Unfortunately, this presents complications when dealing with VsNetClient and VsNetServer modules.

Unlike other VsEntity subclasses, the "start" and "stop" methods for VsRemote do not call the "start" and "stop" methods for any of its children that are VsNetClient or VsNetServer modules. Instead, it evaluates the callback for these module with a "-start" or "-stop" argument. This lets its VsNetClient and VsNetServer children know that a "start" or "stop" was issued without actually calling their "start" or "stop" methods.

4.2.5 Remote Sources

Remote sources are implemented through the interaction of Tcl code in the VsVideoSource and VsAudioSource modules, the SetupRemote procedure, and the vsremotedevice Tcl script. The SetupRemote procedure is a Tcl procedure used by the local side of an application to set up remote sources, remote filters, and remote sinks. The vsremotedevice script is a Tcl script that sets up the remote environment and underlying remote modules used to implement remote sources, remote filters, and remote sinks.

For simplicity, only the implementation of remote video sources will be described. However, remote audio sources are implemented in exactly the same fashion.[7]

Local Source/Sinks Stubs

Say that an application wants to access source Z available only to host Y. In order to create a video source module, the "create" method (consisting of Tcl code) for the video source module is run. This procedure parses the name of the desired video source and decides if it is local or remote. If the source is local, the create procedure continues and creates the local source, as in a traditional VuSystem application. If the source is remote, the video source module calls the SetupRemote procedure, telling it the desired source. This procedure proceeds to set up a local stub that will appear to the user to be the remote source.

The SetupRemote Procedure

The SetupRemote procedure first determines the name of the application's display. Next, it creates a VsRemote object, which serves as a local stub for whatever remote device is being created. The procedure then starts one VsTcpListener and one VsNetListener module on available ports. Then, it uses *rsh*[8] to remotely execute the vsremotedevice script on host Y, passing as arguments the port numbers that the listener modules are listening at, the name of the local host, the type of remote device desired (a video source in this example), the name of the device desired (the video sourced "Y:Z" in this example), and any arguments to pass on to the remote device itself. The SetupRemote procedure then uses its listener modules' "wait" commands to block itself until either the client modules on the remote site establish connections to the local listener module or the timeout expires and an error message is reported.
Because Tcl allows scripts to execute UNIX commands, *rsh* is a powerful tool for running processes on remote machines. Combined with a network file system, *rsh* allows VuSystem applications to run VuSystem scripts on any machine in a network. Remote scripts started in this fashion can then use VsTcpClient and VsNetClient modules to establish in-band and out-of-band connections back to the VsTcpListener and VsTcpListener modules on the local host. VuDP does not need to do any explicit authentication for remote processing because authentication is done within *rsh*.

The vsremotedevice Script

Remote devices are controlled by the vsremotedevice script that runs on the remote machine. This script runs with its own Tcl interpreter in its own name

[7]Simply replace "video" with "audio" in the text below to produce a description of how remote audio sources are implemented.

[8]rsh is a standard Berkeley UNIX command used to execute a command on a remote host.

space, and communicates with the rest of the application only through explicit communication channels.

When the vsremotedevice script is executed, it first initializes the X toolkit, creates an application context and a display object to work with, and initializes the VuSystem. This creates a new VuSystem Tcl interpreter with its own name space on the remote host. The script then creates VsTcpClient and VsNetClient modules that connect to the VsTcpListener and VsNetListener modules started by SetupRemote. This establishes the in-band and out-of-band connections. Next, the vsremotedevice script creates the appropriate local device module (in this case a local video source for video source Z.) The output of the video source is connected to the input of the Tcp client module, so that the media stream produced by the video source is automatically sent over the network to the application.

Finally, the vsremotedevice script sets up the appropriate callbacks for the VsNetClient module so that any commands sent to it are automatically forwarded to the actual video source module and any results from these commands are automatically sent back over the out-of-band connection to application A. Because VuSystem callbacks are executed asynchronously and because there are no guarantees on ordering, the vsremotedevice script is required to reorder the callbacks it receives so that the commands passed on to the local video source module are give in the same order that they were sent by application A. Once these callbacks are set up, the vsremotedevice script loops and devotes itself to event-processing. When the vsremotedevice script senses that either the in-band or out-of-band connection has been broken, it destroys all of its modules and exits.

While the vsremotedevice script is getting started, the SetupRemote procedure on the local side uses the "wait" command on its VsNetListener and VsTcpListener modules to block until the remote VsNetClient and VsTcpClient modules establish the out-of-band and in-band connections, respectively. Once the connections are made, SetupRemote destroys the listener modules. It then aliases the output of the VsTcpServer module to be the output of the VsRemote module. Then, SetupRemote overrides the "unknown" routine of the VsRemote stub with a routine which uses the out-of-band connection to automatically send any unknown commands over the network to the vsremotescript on the other side. (Every VuSystem subclass of VsEntity has an "unknown" routine which is invoked when an invalid command is executed on it. Typically an error message is produced, but in this case the command is simply relayed over the net to the remote device the command was destined for.) Finally, the VsRemote module is returned as the result of creating the remote source module.

To illustrate the setup of a remote source, first application A on host X creates a VsSource module, and specifies "-videoSource Y:vidboard1" as the video source. Host X does not have access to vidboard 1, but host Y does, so remote sourcing is used. In figure 4.9, SetupRemote creates a VsRemote module, creates two listeners, and uses *rsh* to start a vsremotedevice script on host Y, passing the appropriate arguments. Next, in figure 4.10 the vsremotedevice script on Y creates the appropriate local source and creates client modules that connect to the listeners. To finish setting up the remote source the SetupRemote procedure sets up a callback for its VsNetServer module that will relay "start" and "stop" messages over the out-of-band connection without actually starting or stopping the server module. SetupRemote then returns the VsRemote object as the stub representing the video source. Media flow then begins.

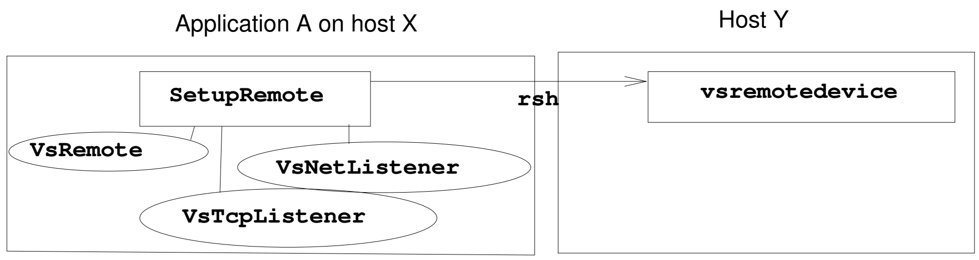

Figure 4.9: Setup Remote Called, Remote Device Script Started

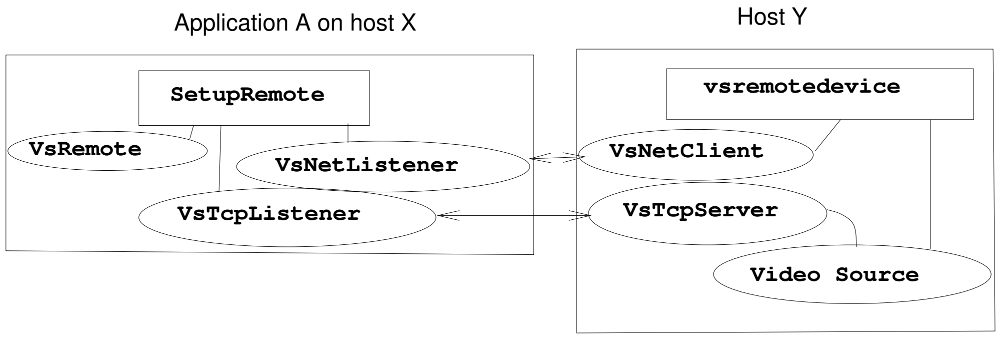

Figure 4.10: Remote Source and Connections Setup

Starting and Stopping

The VsRemote routine has unique start and stop behavior in that when it is started or stopped it does not start or stop any of its children that are VsNetClient or VsNetServer modules. Instead, it executes callbacks for those modules to signal to them that they would have been started or stopped. This is important because of the way that the local stub and the remote source module are connected via the in-band and out-of-band connections.

The VsNetServer module should not be stopped when a “stop” is issued to the local VsRemote stub because the out-of-band connection is necessary to transfer any reconfiguration commands that may occur after the “stop”. Sources are normally stopped before reconfiguration and then restarted, but if the VsRemote stub stops its VsNetServer child module then the out-of-band connection will be broken and no reconfiguration information can be passed to the remote source. Instead, starting or stopping the local stub causes a “start” or “stop” message to be relayed by the VsNetServer module to the underlying remote source module. This starts or stops the underlying remote source without breaking the out-of-band connection used to implement the remote source.
Note that it is not necessary to intercept the “start” and “stop” commands for the VsTcp modules, because VsTcp modules stop media flow but remain connected after being stopped. Hence, the standard VuSystem starting and stopping mechanisms suffice.

Transparent Distribution

Because of the way that remote sources are set up, the VsRemote object returned to the controlling application will appear to be a local video source module. By overriding the appropriate procedures and using the “unknown” mechanism for relaying commands, any commands for the remote source (either commands in the application script or commands entered in the interactive programming environment) will be automatically forwarded by the stub to the vsremotedevice script controlling the remote source. This script then automatically forwards the commands to the source module itself, which handles them and returns a result, which is then forwarded by the script back to the VsRemote stub. The stub then returns this result as the result of the command. Hence, any commands applied to the stub will have the same effect and return the same result as if they were applied directly to the primitive source module. This causes the stub to appear to the user to be a local source module.

In-band Connections

The output of the VsTcpServer module created by SetupRemote is aliased as the output of the stub, and the input of the VsTcpClient module is connected to the output of the actual source module in the vsremote device script, so the media that flows out of the actual remote source module on host Y is automatically routed over the network and appears to be flowing out of the output of the stub module in application A on host X. The output of the stub can be reconnected just like the output of any source, and so as far as media flow is concerned the remote source looks just like a local source.[9]

[9] Note that piping the media over a slow network may cause remote sources to appear to be slower than local sources.

The Control Panel

As described previously, the control panel for a remote source (or sink) appears in its own window, separate from the main control panel for the application, but otherwise identical to the control panel for a local source. To implement this, X-Windows is exploited. When a control panel command is seen by the stub, it relays it to the vsremotedevice script. The script has an X-Windows display variable which names the display of the application that connected to it. When the vsremotedevice script is relayed a control panel command, it creates a new window on that display, and arranges for the remote source to draw its control panel inside that window. The callbacks for controls drawn by the remote device are directly connected to the actual remote device, so the control panel works as it does for local devices.

For the example, say that application A on host X is using display N. Then, the vsremotedevice script on host Y which controls source Z will also use display N. When the user of application A clicks the control panel button, the remote source stub in application A will forward the control panel command to the vsremotedevice script on host Y. The script will then create a new window on display N, and pass this widget to the control panel mechanism for source Z. Source Z will then draw its controls inside this window on display N. The user of application A will thus see two control panels: one containing the controls for the remote source Z, and one containing all other controls.

4.2.6 Remote Sinks

Remote sinks are implemented in the same way as remote sources. First, the "create" routine of the sink module parses the name of the sink to determine if it is local or remote. If it is remote, it calls SetupRemote with the appropriate arguments. SetupRemote in turn creates the VsTcp and VsNet modules as for sources and starts a vsremotedevice script on the appropriate machine. The vsremotedevice script creates VsNetClient and VsTcpClient modules to set up the in-band and out-of-band connections and creates the appropriate local sink module. It then connects the VsTcpClient module appropriately, and sets up the mechanisms to forward commands to and return results from the sink module in the same way that it does for a remote source. SetupRemote then destroys the listeners, does the appropriate aliasing and overriding of the "unknown" routine to make the distribution transparent, and returns a VsRemote stub to masquerade as the sink module.

There is no difference in the way that SetupRemote and the vsremotedevice script handle sources and sinks. In fact, the same script code handles both sources and sinks. In order to use exactly the same code for both sources and sinks, the SetupRemote procedure and vsremotedevice script connect both the input and output ports of the in-band connection. In fact, only the output port needs to be connected for remote sources, and only the input port needs to be connected for remote sinks.

4.2.7 Remote Filters

Class and Instance Procedures

Before discussing the implementation of remote filters, it is useful to briefly review the VuSystem object system. Each module is associated with a class and is typically implemented as a C++ class. But each class is also itself an object in the VuSystem Tcl object system, and so there is a distinction between class methods and instance methods.

Class methods are methods on the class itself; for example, the procedure for creating new instances of a class is a class method. Hence, the line “VsPuzzle vs.vpuzzle” invokes the creation method for the VsPuzzle class object to create a new instance of the VsPuzzle class named vs.vpuzzle.

By contrast, instance methods are methods on instances of a class. VuSystem modules are instances of classes. Examples of instance methods for the VsPuzzle module are “solve”, “scramble”, and “dimension”. Methods that change parameters and do other sorts of reconfiguration are generally instance methods. This distinction between class methods and instance methods is important in the remote filter implementation.

The VsRemoteFilter Module

Remote filters are based on the VsRemoteFilter module, which is a new class implemented entirely in Tcl. The VsRemoteFilter module is very much like the VsVideoSource or VsVideoSink module, in that it is a sort of “wrapper” that is put over the lower-level modules. Most of the remote filter implementation is very similar to the remote source and remote sink implementations.

When a new VsRemoteFilter module is created, it first checks the name of the host on which the filter is to be placed. If it is the local host, then the filter is created locally. To do this, a local filter of the appropriate type is created and made a child of the VsRemoteFilter module. The VsRemoteFilter module itself then advertises all the methods of the underlying filter, so that when these methods are invoked on the VsRemoteFilter module they will automatically be executed on the underlying filter module. For example, say that a VsRemoteFilter module is created that will instantiate a VsPuzzle filter on the local machine. First, a new VsPuzzle module is created and made a child of the VsRemoteFilter module. Then, the various VsPuzzle methods - such as the commands to change the dimension, to scramble the puzzle, and to solve the puzzle - are advertised by the VsRemoteFilter module, which means that any time one of these methods is invoked on the VsRemoteFilter

module it is automatically invoked on the underlying VsPuzzle module.[10]
If the VsRemoteFilter is to be placed on a remote host, it calls the SetupRemote routine with the appropriate arguments. SetupRemote in turns creates the VsTcp and VsNet modules to handle the connection, and uses *rsh* to start a vsremotedevice script on the appropriate remote machine. This remote device script then creates the filter on the remote machine and sets up in-band and out-of-band connections back to the VsTcp and VsNet modules in the SetupRemote routine. SetupRemote then destroys the listener modules, connects the VsTcp module appropriately, and sets up the mechanisms by which filter methods applied to the local stub are automatically forwarded to (and results returned from) the actual remote filter. Finally, SetupRemote performs the appropriate aliasing to make the local stub appear to be an actual filter module, and returns a VsRemote stub to act as a filter module.
The creation and setup of a remote filter is shown in figures 4.11 and 4.12. First, application A on host X has a local video source and a local video sink and wants to create a remote VsPuzzle filter. In figure 4.11, the VsRemoteFilter calls SetupRemote, which sets up the local listeners and uses *rsh* to start a vsremotedevice script on host Y. In figure 4.12, the vsremotedevice script sets up the remote puzzle module and the in-band and out-of-band communication channels. The application then connects the output of the source to the input of the filter and connects the output of the the filter to the input of the sink. Media then begins to flow from the source into the VsTcpServer module on host X, over the network to the VsTcpClient module on host Y, through the remote VsPuzzle module, back into the VsTcpClient, back across the network, back through the VsTcpServer, and into the sink.

Control Panel Implementation

The control panel for a remote filter appears in its own window, separate from all the controls for local modules. The control panel mechanism is implemented within the VsRemoteFilter "panel" instance method, which is called each time the control panel is opened. When called, the "panel" method first checks to see if the filter is actually local or remote. If the filter is local, it has the underlying filter module draw its controls on the control panel, and then appends the "host" button to these controls. If the filter is remote, the method forwards the "panel" command to the remote filter, telling it to use X-Windows to draw its control panel on the appropriate screen.

[10]There are a few methods - such as the "class" and "panel" methods - that can not or should not be overridden by the methods of the underlying filter.

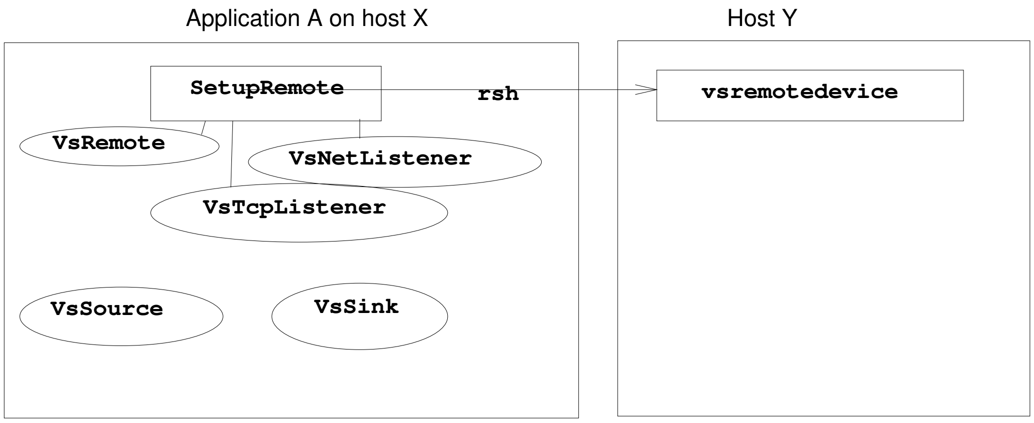

Figure 4.11: Setup Remote Called and Remote Device Script Started

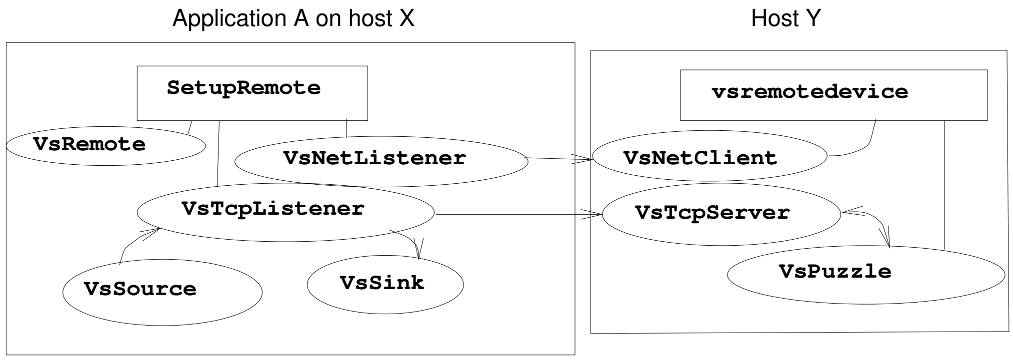

Figure 4.12: Remote Filter and Connections Setup

Chapter 5

Advanced VuDP Applications

The core VuDP tools support the distribution of remote source, remote filter, and remote sink modules. VuDP also provides tools for the development of advanced applications with its general remote evaluation mechanism and the VuDP remote services daemon. VuDP remote evaluation allows programmers to explicitly manage and take advantage of multiple execution environments on multiple hosts, while the VuDP daemon is a high level server that exports useful VuSystem distributed programming services.

VuDP has been used to develop several applications that demonstrate the power and flexibilty of VuSystem distributed programming. These applications can be roughly divided into two groups. First, there are single applications whose component modules are divided among several hosts. Remote sources, remote filters, remote sinks, and remote evaluation are tools for creating these kinds of applications. VsMultiCast is an example of such an application. Second, there are cooperative applications involving multiple users on multiple hosts. The VuDP daemon is designed to provide support for these multi-user applications. VsPigeon, VsTalk, and VsChat are examples of applications in this second group.

5.1 Remote Evaluation

Since applications may wish to do more with distributed programming than create remote sources, filters, and sinks, VuDP was used to create a general remote evaluation facility that allows an application to set up remote execution environments and then evaluate sections of Tcl code in these remote environments. This is an extremely useful mechanism because VuSystem applications consist of Tcl scripts (the out-of-band code) which set up, configure, and manipulate in-band modules (typically written in C++.) Hence, VuDP remote evaluation allows an application to set up remote execution environments and then execute arbitrary sections of out-of-band code in these remote execution environments. VuDP applications can explicitly operate in multiple environments on multiple hosts by specifying in which environment out-of-band Tcl code is to be executed. (Of course, for simplicity and backwards compatibility the default is the local execution environment.) To allow for direct interaction between modules in different environments, VuDP programmers can use the VsTcp and VsNet modules to set up in-band and out-of-band communication channels between peer execution environments.

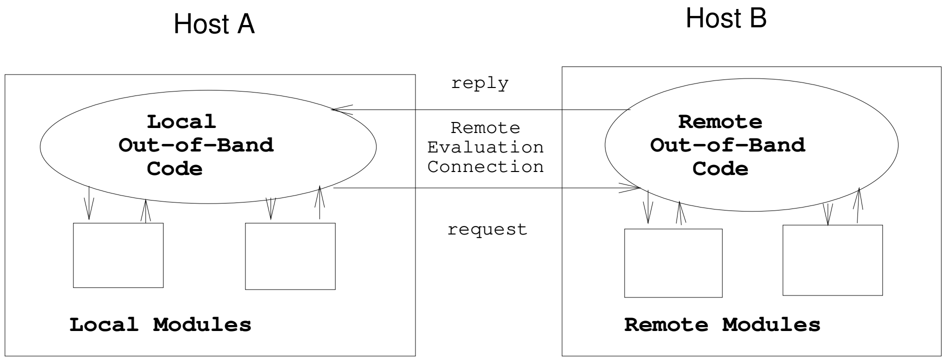

Figure 5.1: VuDP Remote Evaluation

5.1.1 The Remote Evaluation Interface

VuDP applications can set up remote execution environments in which to perform remote evaluation. Each such environment must be explicitly created (via a VsREV module). Each remote execution environment contains its own Tcl interpreter and name space and maintains state across remote evaluation requests. The interpreter and corresponding environment are destroyed when the connection is closed. All details of authentication and data marshalling are taken care of by the implementation.

It is important to explicitly define the programmer's view of the remote interpreter and execution environment. In VuDP, the programmer sees a remote interpreter as being completely separate from the local interpreter. Each remote interpreter has its own name space and shares no state with either the local interpreter or any other remote interpreter. This means that, in general, variables, objects, and procedures in one interpreter are not available to peer interpreters. For example, say that a local application uses a VsREV module to establish a "remote" evaluation environment. The local and remote environments would contain separate Tcl interpreters and name spaces, and the variables, objects, and procedures that exist in the local interpreter would not exist in the remote interpreter (and vice-versa).

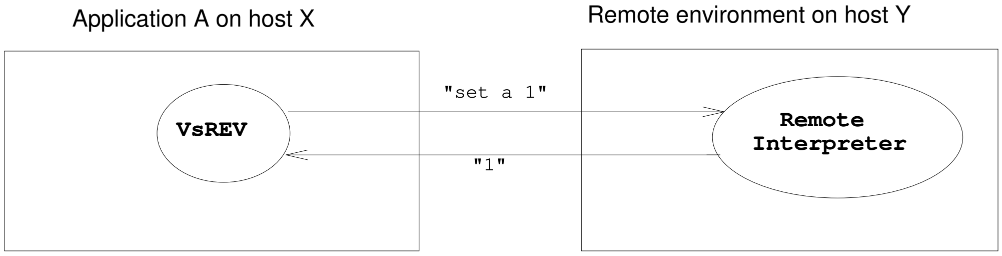

Figure 5.2: Example of Remote Evaluation

Though this may seem somewhat constraining, there are two good reasons for completely segregated name spaces. First, it provides the programmer with a simple, easily understandable relationship between the local and remote sections of the application. There is no need for the programmer to worry about subtle, poorly understood interactions between the local and remote environments, because the only interactions that occur are exactly those that the programmer explicitly sets up. Second, the segregation model presents the programmer with a model that closely resembles the underlying implementation. This avoids hiding expressive power, provides good performance, and allows programmers flexibility in deciding how to use the distributed programming tools.

One effect of segregating local and remote name spaces is that modules existing in separate name spaces can interact only through explicit communication channels. Any in-band communication between local and remote interpreters must pass through VsTcp modules, and any out-of-band communication between local and remote interpreters must pass through VsNet modules; generally no other interactions between remote modules are allowed.[1] Because all simple Tcl variables are strings, data marshalling is not an issue, and so variable values are easily passed between interpreters.

Finally, an important issue is deciding how much state to place in remote interpreters when they are first created. A certain amount of "boilerplate" code exists in each VuSystem application script to initialize the X-windows interface, etc. In order to initialize the VuSystem and allow for creation of remote modules, a remote interpreter must initialize the X toolkit and create an X-Windows application context, display, and top level widget. Only this bare minimum of initialization and setup is done when a remote interpreter is created.

[1]One notable exception is using a network file system to allow remote interpreters to interact through shared files.

Discussion

The VuDP view of remote evaluation is that remote interpreters exist in completely segregated execution environments. Other choices are to have all peer interpreters appear to run in the same execution environment, or to have peer interpreters run in separate environments that implicitly share some proper subset of their state. Having local and remote evaluation occur in the same environment makes little sense. If remote evaluation occurs in exactly the same environment as local evaluation, then improved performance (via parallelism or access to a remote machine that is faster than the local machine) is the main reason for using remote evaluation. And even this theoretical performance improvement is of dubious value, for several reasons. Even if no asynchronous execution is allowed, there must be a good deal of communication between the local and remote interpreters to keep their states consistent. One could imagine either sending the entire state with each REV request, or having the local and remote interpreters dynamically inform each other of state changes.

If asynchronous execution is allowed and peer interpreters appear to run in the same execution environment, then it becomes extremely difficult to maintain the local and remote interpreters in consistent states. Doing so requires a tremendous amount of locking, synchronization, and intermachine communication to keep the local and remote interpreters consistent. Even if the VuSystem did allow for multi-threaded applications (which it doesn't), maintaining shared state between the local and remote interpreters would complicate the system and most likely negate some or all of the performance improvement achieved by allowing remote evaluation.

Another approach is to specify some proper subset of interpreter state which is shared between local and remote interpreters. However, there are many problems with any kind of implicitly shared state.

- Shared state must somehow be marshalled and transmitted transparently.
- In the VuSystem, where applications are Tcl scripts running on UNIX systems, it is difficult or impossible to transmit certain types of state, such as UNIX file handles.
- Shared state is subject to the consistency problems described above, which only become much worse when asynchronous execution is allowed.
- Application developers must be familiar with all the intricacies of partially shared state, which makes the system more difficult to use.

Finally, any shared state between local and remote interpreters must be reflected in the VuSystem runtime programming environment. With segregated interpreters, the best way to handle the runtime programming environment is to separate the representations of the local and remote interpreters to reflect their independent natures. (This is described in more detail later in this section.) By contrast, there seems to be no good intuitive runtime programming representation for local and remote interpreters with shared state.

5.1.2 Using VuDP Remote Evaluation

VuSystem applications can set up remote execution environments by using the VsREV module. Upon creation, a VsREV module takes its host argument and creates a new execution environment and a new interpreter in the new environment on the specified host. A VuSystem application can maintain an arbitrary number of different remote evaluation connections to an arbitrary set of hosts via multiple VsREV modules. Each VsREV module is responsible for maintaining the connection to one remote execution environment and interpreter.

Handling Remote Evaluation

Once a connection is set up using the VsREV module, remote evaluation can be performed by using the VsREV "rev", "revsend", and "flush" commands. The "rev" command takes as an argument a Tcl command to execute. It sends this command to the remote interpreter, evaluates it on the remote interpreter, and returns the result of the remote evaluation as its result. The application making the "rev" call is blocked until either the return value is received or a timeout expires.

If no return value is desired, then the asynchronous "revsend" command can be used to send the remote evaluation request in the background, without blocking the local application. In order to make sure that previous remote evaluation requests sent with "revsend" have been sent, the VsREV "flush" command can be used to block the local application and immediately send all previous "revsend" commands that are waiting in the background.

Note that a "rev" command does not automatically flush previously buffered "revsend" commands. A "flush" should be used between "revsend" and "rev" commands if the "rev" commands depend on previous "revsend" commands.

If asynchronous input from the remote interpreter is desired, a callback function can be specified for the VsREV module which will be executed each time asynchronous input is received from the remote interpreter. When data arrives via the callback, the callback routine will be called with the arguments "-command", whose value contains the data received, "-call", which is set to 1 if the remote interpreter is blocking and waiting for a reply (and set to 0 otherwise), "-num", the sequence number of the message, and "-eof", which is set to 1 if the connection has been closed by the remote interpreter.

All remote evaluation is done with the privileges of the user of the local application. Remote evaluation can do no more and no less than what would be possible to do by simply logging into the remote machine.

A remote evaluation connection is terminated when the VsREV module is destroyed. When the remote interpreter senses that the connection has been broken, it destroys itself.

An Example of a Remote Evaluation Connection

An application wants to set up a remote evaluation connection to host Y. First, the application creates a VsREV module named "vs.rev", and configures it to connect to host Y. Then, in figure 5.3, the application starts the module, which creates a remote interpreter running on host Y. The application can then use the VsREV module's"rev" and "revsend" commands to execute Tcl code on the new interpreter on host Y.

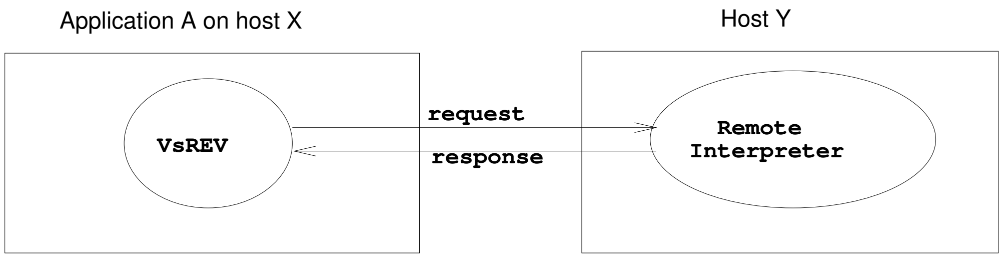

Figure 5.3: Remote Evaluation Connection Established

The code fragment in figure 5.4 illustrates examples of using "rev" and "revsend". The application sets variable "a" to be 10 in the local environment and 1 in the remote environment, sets variable "b" to be 5 in the local environment, sets variable "c" in the remote environment to have the value of variable "b" in the local environment, sets "d" in the remote environment to have the value of "a" in the remote environment, flushes the sends[2], uses "rev" to execute "set c" and "set a" in the remote environment to find out the values of "c" and "a" in the remote environment, and finally runs "set a" to find the value of a in the local environment. In the code fragment, the lines beginning with ":" represent the return value of the line above. Note that "revsend" does not return a value, but "rev" does.

The connection is broken when the VsREV module is destroyed, causing the remote interpreter and all its associated state to be destroyed.

[2]The use of "flush" is described below.

VuDP Interactive Programming

VuDP provides one interactive programming environment for each interpreter the application has access to. In other words, two windows (one graphical representation window and one textual interface window) are displayed for the local section of the application, and two windows are displayed for each remote evaluation connection. The local interactive programming window provides access to all local modules (including any VsREV modules used to establish remote evaluation connections), and provides a graphical and textual interface to the local interpreter. Each remote evaluation partition has its own windows which provide access to all the modules that have been created for that interpreter.[3]
Having separate interactive windows for each remote evaluation connection reflects the separate nature of the remote execution environments.

Discussion

There are several reasons why it does not make sense to integrate the local and remote interactive programming environments into a single set of windows. First, consider the top interactive programming window, containing the graphical representation of the application. Remote evaluation occurs in a separate environment and local and remote modules can not directly interact. A second reason not to display local and remote modules in the same set of windows is that modules in separate execution environments can not be directly connected together using the graphical programming tools. Hence, displaying them in the same window may frustrate users who are not able to use the graphical programming tools to interconnect them. Finally, if local and remote modules are presented in the same set of windows then there must be some mechanism for marking remote modules with the host where they exist. This labeling of modules is relatively straightforward if an application is limited to only one remote evaluation connection to any host, but becomes much more complicated and impractical when (as is allowed by VuDP) applications are allowed to have multiple remote evaluation connections to different address spaces within the same host.
If local and remote environments are accessible through the same window, then how does a user specify which interpreter is to be used to evaluate dynamically entered commands? This problem is even more complicated than it first appears because VuDP allows applications to maintain an arbitrary number of remote evaluation connections to an arbitrary set of hosts. Forcing a user to specify which interpreter to use for each request would unnecessarily complicate both the interactive programming implementation and the use of VuDP.

[3] Because directly accessible remote modules are only created via remote evaluation, the interactive programming interface is only an issue when remote evaluation connections are established. This interface is not an issue for transparent distribution - such as is done with remote sources and remote sinks - since transparently distributed modules should appear to be local.

VuDP's multiple-environments approach solves all of the above problems, and has the feature that the interactive programming windows parallel the separate remote interpreters.

5.1.3 Implementation

The VuDP remote evaluation mechanisms are implemented entirely in Tcl; no new low-level C++ code was necessary. The implementation consists of two parts: the VsREV module and the vsremoteeval script.

The VsREV Module

If application A on host X wants to open a remote evaluation connection to host Y, it simply creates a VsREV module configured to connect to host Y. This sets up the remote evaluation connection to host Y. Once the connection has been established, the VsREV module's "rev", "revsend", and "flush" routines can be used to perform remote evaluation.

Upon creation, the VsREV module sets up a VsNetListener module on some (non-default) port and figures out which display application A is using. The module then uses "rsh" to start up the "vsremoteeval" script on host Y, passing as parameters the name of the local host (host X), the name of the display application A is using, and the port number at which the VsNetListener is listening. The VsREV module then uses the VsNetListener's "wait" command to wait for the vsremoteeval script on host Y to connect back to the VsNetListener module.

The vsremoteeval Script

The vsremoteeval script controls the creation and maintenance of the remote interpreter and the remote execution environment. It is invoked via 'rsh' by the VsREV module and runs on the machine to which the remote evaluation connection is being made; for the above example, the vsremoteeval script runs on host Y.

The script first initializes the X toolkit, creates an application context, and opens the appropriate display. Next, vsremoteeval initializes the VuSystem and creates a VsNetClient module configured to connect to host X on the port specified by its arguments. The VsNetClient module is then started, setting up an out-of-band connection between the local VsREV module and the remote execution environment of the vsremoteeval script.

The Command Callback

Finally, the vsremoteeval script sets up a command callback routine to handle input that the script receives over the out-of-band connection. The command callback routine extracts the parameters of incoming commands to determine the sequence number of the command, whether or not the connection has been broken (as signaled by an end-of-file message), whether or not the remote application is blocking and waiting for a response, and the body of the command. Incoming messages are placed in a queue tagged with their sequence number, and an interpreter routine interprets the newly received messages in the order of their sequence numbers.

The Remote Evaluation Interpreter Routine

The remote evaluation interpreter routine must use the sequence numbers of incoming remote evaluation commands to reorder them so that they are evaluated in the same order that they were sent.[4] When a new command arrives, the interpreter scans the queue of incoming messages to find the message with the proper sequence number. If it doesn't find it, it does nothing.[5] If the interpreter does find the command with the correct sequence number, it remotely evaluates the command, returns the result (if necessary), and loops back to search for the next command to evaluate.

Commands are evaluated by simply interpreting them in the global environment of the vsremoteeval script's Tcl interpreter. If a return value is expected, the result from evaluating a command is saved and returned back to the calling interpreter. If no return value is expected, no return value is sent.

When the vsremoteeval script receives an end-of-file message over the out-of-band connection, it can determine the sequence number of the last command to be evaluated by decrementing the sequence number of the end-of-file message. When it has received the end-of-file message and knows that it has evaluated the last command, the script quietly destroys the remote execution environment and exits.

Remote Interactive Programming

Opening an interactive programming window causes two windows (one graphical program representation window and one textual interpreter interface window) to appear for each interpreter that an application is using. Since there will always be two windows for the local interpreter, the total number of interactive programming windows will be two plus twice the number of active remote evaluation connections, since each remote evaluation connection is associated with a separate interpreter.

In the standard VuSystem, opening the interactive programming windows involves calling the VsVisualShell procedure on the top-level object for that application. (In VuSystem applications, there is normally a top-level object called "vs" that is an ancestor of all other objects.) This opens the windows, sets up the graphical

[4] This is because Tcl does not preserve the ordering of callbacks.

[5] This will happen if the callbacks are reordered and the message that just arrived has a higher sequence number than the one the interpreter wants and so should be temporarily enqueued. Messages will not be lost during transmission because out-of-band connections use the TCP protocol.

programming tools and the textual interface to the interpreter, and then recursively calls the "drawNodes" procedure on all of its children. This causes each of its children to draw representations of themselves (typically boxes labeled with the appropriate module type) in the graphical window and draw connections between modules where media flows, hence producing a graphical representation of the application.

The remote interactive programming window is implemented in a very straight-forward fashion. When the "drawNodes" procedure for the VsREV module is called, it draws itself as if it were a local module, but before exiting it sends a message to its remote interpreter telling it that the program button has been pushed. The command interpreter for the vsremote script has a special case for receiving a program button command, and instead of passing this command on to be evaluated (like a normal remote evaluation command) it instead calls the VsVisualShell procedure on the top-level object for the remote interpreter. Hence, the remote interpreter displays an interactive programming window (just as if it were the main local interpreter) by drawing a window in its display, setting up the textual interface and the graphical programming tools, and calling the "drawNodes" procedure for all its modules. Because the X display for remote interpreters is set to be the same as the display for the local application, all remote programming windows - along with the local interactive programming window - will appear on the user's display.

Once started, the programming windows for remote interpreters are no different than those for local interpreters. Remote objects can be created, reconfigured, and destroyed using the graphical tools, and commands can be directly entered into remote interpreters using the textual interface. Remote programming windows persist until explicitly closed; they are not automatically destroyed when the local programming window is closed.

Error Reporting

Note that the display for the remote script is the same as the display for the local application that started the remote evaluation connection. Hence, if an error occurs while performing remote evaluation, the normal VuSystem error window is automatically popped up on the display that the application is running on. This is powerful, convenient, and clean.

5.2 The VuDP Daemon

Though remote evaluation is a general and powerful mechanism for distributed programming, it is also desirable for applications to have access to a higher-level server which exports a set of useful distributed programming services. This kind of server is especially useful for applications involving multiple users. The VuDP daemon is an application that runs on every machine where at least one person is logged in. The daemon listens at a well-known port, and client applications running on any machine can connect to the daemon to request services. The daemon exports a certain set of services to allow remote applications to access the resources of the daemon's local machine and to facilitate synchronization between users in multi-user applications.

5.2.1 The VuSystem Daemon and Remote Services

Starting the VuDP Daemon

Because the VuDP daemon exists primarily for multi-user applications, a daemon is typically started in a user's startup file when he/she logs into a machine. This has many advantages. First, the user can decide whether or not the daemon is running; by starting a daemon a user is basically announcing that he wants to participate in multi-user VuSystem applications. Second, it allows users to customize their daemons to offer exactly the set of services that they want to export to other applications connecting to their daemon. For example, a user may decide to customize his daemon to disable the VuDP messaging system. Third, it allows a single daemon process to handle all VuDP daemon services for a single host. This means that the overhead of process creation and VuSystem initialization required to start a new daemon is incurred only once, at log-in time. Finally, if a single VuDP daemon handles all daemon service requests for a single machine, then all those requests are handled in the same address space. This provides the potential to use the daemon's address space as a means of communication between remote VuSystem applications. Because the daemon is designed for use in multi-user applications, running the VuDP daemon as a user process is not a problem, as there should be no need to access daemon services on a host with no users.

Alternatives to per-user Daemons

The main alternative to having the daemon started when a user logs into a machine is to use inetd[8] to start a new daemon on a host each time a connection to a daemon on that host is requested. This has the advantage of always having a daemon available on every host, even if no user is logged in. On the other hand, VuDP remote evaluation can always be used to access any machine, whether or not anyone is logged in, so it isn't necessary to always have a daemon running on every machine. And, having a daemon started by inetd also does not allow for per-user customization of the daemon, and gives users little control over the VuDP daemons running on their machines. Further, the overhead of process creation and VuSystem initialization - a non-trivial amount of computation - is incurred each time a new daemon connection is established. Having to create a new daemon for each connection would significantly impact performance, especially for simple service requests such as made by the VsPigeon messaging application. Finally, creating a separate daemon process for each connection would disallow using the daemon's address space for any potential communication between remote applications connected to the same daemon.

Security

With the VuDP daemon, security is not a problem because the daemon exports an explicit set of services. Hence, client applications can only do on the remote machine exactly what the daemon lets them do. As long as the daemon services are designed with security in mind, the daemon should not introduce any security loopholes.

The daemon services and remote evaluation mechanism complement each other without creating any security risks. Daemons generally have access to all resources of the machine they run on, but only allow access to those resources in an controlled fashion. By contrast, remote evaluation allows an application access to exactly the remote resources that the user of the application could access by logging into the remote machine.

5.2.2 Using the VuDP Daemon

VuSystem applications can use the VsNetClient module to connect to a VuDP daemon on a remote machine to request daemon services.

Connecting to the VuDP Daemon

For an VuSystem application to connect to a VuDP daemon on a remote machine, it must first create a VsNetClient module and configure it to connect to the appropriate host. The default port for the VsNetClient to connect to is the port that the VuSystem daemon listens to, so if a VsNetClient does not explicitly reconfigure its port then it will automatically be set up to connect to the VuDP daemon.

Handling a VuDP Daemon Connection

Once a connection is established, an application can access the daemon resources through the VsNetClient "send" and "call" commands. Depending on the semantics of the daemon command, a "send" command may involve having the daemon return a value; however, the return value will be received by the asynchronous input callback mechanism for the VsNetClient module. By contrast, "call" will always return a value. For both "send" and "call", any errors encountered by the daemon while servicing a request will be returned back to the application.

5.2.3 Implementation

The VuDP daemon is implemented as a single Tcl script, much like any traditional VuSystem application. As discussed previously, the daemon is started when a user logs into a machine. Only one daemon will be started at any time even if there are multiple users on a machine. Only the user that started a daemon can control and reconfigure it.

When a daemon is started, it first creates a VsNetListener and starts it listening on the VuDP daemon port. (This port is the default port for the VsNetListener and VsNetClient modules.) The daemon sets up a connection callback routine to be executed whenever a new connection is made. It then defines a command interpreter routine that will be used by all connections to interpret commands sent to the daemon and dispatch to a routine to provide the appropriate service.

The connection callback is executed when a new connection has been established. Arguments to the callback inform the daemon of the name of the VsNetServer module that the VsNetListener created to handle the connection. The callback routine itself initializes some state for the connection, and then sets up a command callback routine which is executed whenever new input is present for that connection. It then sets up a configuration callback routine that is called when the connection is broken.

The Interpreter

The daemon has a single command interpreter routine that is shared by all connections, though each invocation of the interpreter is for a specific connection. In response to a new message, the routine is invoked for the connection on which the message arrived after the command callback routine for that connection is executed.

The interpreter routine takes a single argument, the name of some VsNetServer object. When invoked, the routine first searches the message queue for the appropriate VsNetServer to try to find the command with a sequence number one greater than the last command handled for that server. (For example, if the last command serviced for some VsNetServer had sequence number 25, then the next time the interpreter routine was invoked for that VsNetServer it would search the server's message queue to find the command with sequence number 26.) If the command with the appropriate sequence number does not exist, the interpreter does nothing and returns. If the appropriate command is in the queue, it extracts it from the queue and dispatches to the appropriate routine to handle the command. When the command has been handled, the interpreter increments the sequence number of the last request serviced for the VsNetServer, and rescans that server's queue of received messages to find the next command to service. The interpreter routine continues to loop in this fashion until it can no longer find the next command in the queue.

Terminating a Connection

When a VsNetServer command callback receives a message with the "-eof" parameter set, it means that the connection has been closed. However, because the VuSystem does not preserver the ordering of callbacks, the callback routine can not destroy the server object and forget about the connection until after it has handled all messages with lower sequence numbers. After all the messages have been handled, the connection is broken, though it may not be desirable to destroy the server object immediately in case it has children that persist after the connection is broken.
Note that VuDP daemon connections will almost always be terminated on the client side. The only time that the daemon will terminate a connection is if the daemon crashes or is otherwise shutdown by the user that started it (such as if the user logs off.)

Command Routines

When a command routine is called, it is passed the arguments to the command. It is also told whether or not the command was executed as a "send" or a "call" so that the command routine knows whether or not it should provide a return value. For example, the VsPigeon command routine parses its arguments to figure out the sender's name and the message to display. It then displays a window on the screen of its local host containing the message, the sender's name, and the current date and time. Finally, it sends an acknowledgement back to let the VsPigeon script on the other side of the connection know that it has successfully displayed the message.

Error Reporting

If an error occurs while handling a VuDP daemon command, an error message is returned over the connection. The error message returned is the error message received by the daemon, prefixed by "ERROR:". Note that only errors encountered while attempting to interpret commands are forwarded across connections; errors resulting from the daemon's internal workings are reported to the screen of the machine running the daemon.

5.3 VsPigeon

VsPigeon is an application which can be used to send messages (via pop-up windows) to remote machines. It is a good example of an application that is simple to write using the VuDP daemon but that would be difficult or impossible to write in the traditional VuSystem. An example of a VsPigeon message is illustrated in figure 5.5.

To send a message, a user starts VsPigeon and supplies the name of the host machine to send the message to, an optional "from" argument, and a message. If input is routed into VsPigeon, it is automatically sent as the body of the message. If no input is routed into VsPigeon and there is a message specified on the command line, that message will be sent as the message body. If no input is routed into VsPigeon and there is no message specified on the command line, then VsPigeon prompts the user to enter the body of the message.
Figure 5.6 illustrates the sending of a VsPigeon message from host X to host Y.

5.3.1 Implementation

VsPigeon is implemented as a Tcl script which takes as input a host to send the message to, an optional "from" argument, and a message to send. When invoked, the script creates a VsNetClient, connects to the VuDP daemon on the appropriate machine, and sends a message asking the daemon to display the message. Because the VuDP daemon has access to the display of the machine it is running on, it can display the message.

5.4 VsTalk

VsTalk is a VuSystem application that allows for real-time visual interaction between two users on different machines by having each user choose a video source to display to the other user. During a VsTalk session, both users are presented with two video windows: one window displays the video stream chosen by that user, and the other window displays the video stream sent by the other user. This way, both users see both the video that they are sending and the video that they are receiving.
As an example, say that user A on host X wants to start a VsTalk session with user B on host Y. First, user A starts the VsTalk application, specifying that a connection to host Y is desired. Then, user B on host Y is queried via a pop-up window whether he wants a VsTalk session with user A on host X. If B refuses, the application terminates, and user A is informed that the VsTalk request was refused by user B. If B accepts the VsTalk request, then both user A and user B are presented with two live video windows, one displaying user A's video source and the other displaying user B's video source.
Figures 5.8 and 5.9 illustrate user A on host X initiating the session and having the daemon on Y query user B if a session is desired. B responds affirmatively, and the session is established.
VsTalk existed before the VuDP extensions were added, but in order to set up a session it was necessary to have the two users simultaneously running VsTalk in order to start the connection. Hence, unless a VsTalk application was kept running at all times, any time a VsTalk session was desired it was necessary to use some

other medium of communication (such as the telephone, email, or oral communication) to synchronize and agree upon a starting time for a VsTalk session. This made using VsTalk awkward and impractical. But, with the VuDP extensions, the daemon is used so that users are notified via pop-up windows when a VsTalk request is made, and (if the request is accepted) the VsTalk session is automatically initiated.

5.4.1 Implementation

VsTalk is a single TCL script that can run as the "connector" or the "receiver". Running as the connector means that the script will attempt to initiate a VsTalk session; running as the receiver means that another party has attempted to start a VsTalk session, and the script is invoked as the receiver to accept the request and start the session. Thus, VsTalk can be started in one of two modes: "connect" mode or "receive" mode. Whenever it is started by a user, it starts in "connect" mode. Normally only the VuSystem daemon starts VsTalk in "receive" mode.
A "connect" mode VsTalk is started when a user initiates a new VsTalk session. When the other party has been specified, the "connect" mode VsTalk connects to the VuDP daemon on the appropriate machine and has it query the appropriate user whether or not a VsTalk session is desired. If so, the daemon starts a "receive" mode VsTalk on the appropriate machine. The "receive" mode VsTalk then proceeds to set up in-band and out-of-band communications channels to the originating "connect" mode VsTalk, beginning the session.

5.5 VsChat

VsChat is an application that allows an arbitrary number of users to communicate in real-time via a shared chat board. Currently it only allows for textual communication, but the addition of video and audio is a natural extension. VsChat is an interesting use of VuDP because it allows an arbitrary number of users to interact in the same application.
When VsChat is invoked, it displays a chat window on the screen. The window is divided into two sections. The larger top window displays output of text sent to the chat board by all users. The smaller bottom window is used to input text to send to the chat board. Upon joining a chat session, the list of users currently involved in the chat is displayed. Messages can be sent to the chat board by simply entering them at the prompt in the lower window. Messages will be automatically sent to all participants and displayed in the upper window of all active VsChat applications. So that everyone knows where messages originated, displayed messages are prefixed with the username of the sender. Finally, the application displays special messages in the upper window when a user either joins or leaves a VsChat session. An example of the VsChat window is shown in figure 5.10.

VsChat is designed to accommodate an arbitrary number of users. However, since the current version only supports a single chat board, the practicality of its use by tens or hundreds of simultaneous users is limited.
A VsChat session persists until all users have exited the session. If there is no active VsChat session when VsChat is invoked, a new session is initiated.

5.5.1 Implementation

VsChat is implemented as a single Tcl script. This script is responsible for opening the VsChat window, connecting to a current VsChat session if one is active, creating a new session if there is no active chat session, and handling all VsChat interaction.
Each VsChat session consists of one or more participants. One of the participants is designed as the "leader". Details of the responsibilities of the leader will be discussed below.
In an active VsChat session, every participant maintains one out-of-band VsNet connection to every other participant; in other words, the participants are fully connected. These connections are used to both forward messages entered by one user to all participants and to exchange control information. Further, every participant in a VsChat session always has an active VsNetListener module which it uses to accept connections to new participants. Each participant maintains, at all times, a list of four-tuples containing the usernames, hostnames, VsNetListener port numbers, and VsNet module names for all other participants in the session.
Figure 5.11 illustrates the connections maintained during a VsChat session. User A on host X, user B on host Y, and user C on host Z are all participants in the active VsChat session.
When a message is entered by any user in a VsChat session, that message is prefixed with the username of the sender and then copied and sent to all other participants by having the sender loop through all of its connections to other participants to send a copy of the message to everyone in the session. When a participant receives a new message over any of its connections, the message is immediately displayed in the output window.
Note that VsChat sessions persist as long as there are participants. Connections can be established to new participants and terminated when old participants exit an arbitrary number of times. If the "leader" exits the session, then some other participant is designated as the leader and the session continues.

Startup

VsChat is designed to be used as a chatboard for a group of users connected by a local area network. Thus, when a new user wants to join a session, there must be some way to determine if a session already exists, and if so where to go to join it. To do this, a shared file is used. Because a network file system is assumed, this shared file - known as the "chatleader" file - can be accessed by any user on any host.[6] If there is an active VsChat session, the chatleader file contains the hostname and port number at which the VsNetListener module for the leader of the session is listening for new connections. If there is not an active VsChat session, the chatleader file contains "NONE" to indicate that no session is active and hence there is no leader.

When a new participant starts the VsChat script, the script first starts its VsNetListener module at some available port. It then reads the shared chatleader file to determine if a session is active. If there is no active session, then a new VsChat session is set up, the new participant makes himself the leader and writes the name of the host on which it is running and the port number at which the VsNetListener module is listening to in the chatleader file. Upon establishing the new session, the leader is the only participant, so setup is finished and the leader must wait for other users to join the session.

If there is an active session, the new participant reads from the file the host and port at which the session leader's VsNetListener module is listening. It then creates a VsNetClient and connects it to the leader's VsNetListener module to establish a connection. A message is then sent over this connection announcing the new participant's username, hostname, and listener port number to the leader. The leader then adds this information to its list of information about the participants, and returns a copy of the list to the new participant. The new participants creates a VsNetClient module, establishes a new connection to each participant, and announces to each participant that it has joined the session. Each of the other participant's appends the information for the new participant to its copy of the list. Once this has been accomplished, the new participant has joined the session and is announced via a special message displayed on the chat board.

The Participant Information List

At the center of the VsChat implementation is the list of four-tuples containing information about all participants in a session. For simplicity, assume that one participant is participant X, representing user Y running on host Z with the VsNetListener module listening at port N on host Z. Participant X maintains a list of four-tuples containing information about other participants. The first field contains the usernames of all active participants. The second field contains the names of the hosts that the participants are running on. The third field holds the port numbers that the participants' VsNetListener modules are listening at. The fourth field contains the names of the VsNet module (either a VsNetClient or a VsNetServer module) that participant X uses to send messages to and receive messages from the other participants.

[6] VsChat's use of this shared file is a good example of the way that VuDP exploits the capabilities of a network file system.

The 0th index in each list contains the information for the host participant; in the example, the 0th elements of the first three lists for participant X would be "Y" (X's username), "Z" (X's host), and "N" (X's VsNetListener port). The 0th element of the fourth list is a filler.
The list is kept current so that, at any time, each participant has a list containing the correct information for all other participants. They are inaccurate only when some participants have not been informed yet about a new arrival or exit from the session. The leader does not store any more state than any other participant; the only distinction that the leader has is that its hostname and listener port number are stored in the chatleader file.

Adding New Participants

A new participant finds out about the existing participants by connecting to the leader and receiving (from the leader) the list of all existing participants' usernames, hostnames, and listener ports. The new participant then establishes connections to each existing participant and announces that it has joined the session.
To clarify the explanations below, as an example let participant X be a new participant, let participant Y be the leader of the existing session, and let participants A, B, and C be other non-leader participants. First, X creates a VsNetClient module connects it to Y's VsNetListener module. When Y's listener module senses the connection, it creates a VsNetServer to handle the connection, and executes a callback routine. The callback routine first sets up a command callback routine that is executed each time a new message is received over the connection. This command callback tags messages with a sequence number, checks for end-of-file, enqueues the message, and then calls the shared message handling routine. Handling a message may involve informing other participants of a participant entering or leaving a session, or may involve displaying some new text on the chat board.
To continue the example, after initiating the connection Y has set up a command callback routine (with the associated reordering mechanisms) to handle input from X. Because all connections are symmetrical, X must also set up a similar command callback routine for its VsNetClient that will be called whenever X receives input from Y over this connection. All VsNetClient and VsNetServer modules in VsChat have this same sort of command callback routine that is used to accept new messages and reorder them before interpreting them.
Much as for remote evaluation, incoming messages are tagged with sequence numbers so that messages received from any given participant are displayed in the same order that they were sent. Note that this reordering applies only to messages received from a single participant; messages from different participants may be arbitrarily reordered.
The message display procedure takes as its argument the name of the module

controlling the connection to some participant. It then scans the queue of received messages and looks for the one with the appropriate sequence number. If it finds it, it handles the message (typically displaying it on the chat board) and then loops to find the next message, continuing until the next message is not yet in the queue. The routine contains subroutines to handle messages from new participants to leaders requesting information on other participants, messages from new participants to existing participants announcing the new participant, and messages containing text to be displayed on the chat board.

After the connection and the appropriate command callback have been set up, X sends a special message to Y to announce its entry into the session. When Y receives and interprets this message, it adds X's four-tuple to its list. Y then returns a copy of this list to X.

Next, X must establish connections to the other participants. To do this, X creates uses VsNetClient modules to establish an out-of-band connection to each participant in the session. X uses the connections to announces itself to and communicate with the other participants, allowing the other participants add X to their lists. Lastly, X sends a special "X has joined the session" message to the chat board.

Removing Participants

When a non-leader participant leaves a session, it first sends a special message to the board announcing that it is leaving. It then breaks all of its connections and quietly exits. When the leader leaves a session, it first chooses some other participant to become the new leader, and modifies the shared chatleader file accordingly.

As for remote evaluation, the command callback routines for each connection for each participant scan each incoming message for an end-of-file flag. If the flag is set, it means that the connection has been broken. But, since Tcl does not preserve the ordering of callbacks, the connection can not be destroyed until all messages with previous sequence numbers have been received and interpreted.

As an example, say that participants X, Y, and Z are involved in a VsChat session, and X leaves the session. Upon sensing the broken connections, Y and Z are informed via end-of-file messages. Then, when Y and Z have finished interpreting the message from X with sequence number one less than the end-of-file message, the controlling VsNetClient or VsNetServer module can be destroyed. Further, when the last message from X has been handled, Y and Z must remove X's username, hostname, listener port, and associated module name from their lists, since X is no longer part of the session. Once this is done, X has been removed from the session and the session continues.

Sending and Receiving Messages

The VsChat script sets the keyboard focus on the input window, and executes a special callback each time the "return" key is pressed by the user. When this callback executes, it reads in the text entered by the user, and prefixes it with the user's username. Next, it sends this message to be displayed on the output section of the local VsChat window. Finally, it loops through all other participants, and uses the appropriate VsNetClient or VsNetServer module to send the message to all the other participants.

Ending a VsChat session

A VsChat session ends when the last participant exits. Since there must always be a leader, the last participant must necessarily be the leader. Hence, when the leader exits and there are no other participants, the session ends and "NONE" is written into the chatleader file.

5.6 VsMultiCast

VsMultiCast is an application that allows a single video stream to be sent simultaneously to window sinks on the screens of multiple machines. Though it really isn't a multicast[7], it does allow the sharing of a single media stream by several users, something that can not normally be done with the traditional VuSystem.

A VsMultiCast consists of a "master" window and an arbitrary number of "slave" windows. The master window is the one created when the application is started. The master window has controls for adding news hosts to and deleting old hosts from the multicast as well as control panel and interactive programming buttons. The slave windows contain only a screen display and a dismiss button. Examples of VsMultiCast master and slave windows are illustrated in figures 5.12 and 5.13, respectively.

Each time a new hostname is entered in the "Add New Host" box, the master will create a new slave window on the specified host. A slave window is destroyed when either the "dismiss" button on the slave window is pressed or the name of that host is entered in the "Remote Host" box of the master window.

Note that only the master window can control the media source; the slave windows are specifically disallowed from having control panel or remote programming buttons so that there will be no contention for control of the media source.

An arbitrary number of slaves can be added to a VsMultiCast session. However, it is important to note that when a video stream is shared in this manner it is

[7] The VuSystem VsMultiCast is implemented by multiple point-to-point connections. By contrast, the Mbone[16] is used to perform video conferencing over a network that supports multicast addressing.

necessary to do a substantial amount of copying of the underlying media. Further, in the original implementation, shared video source will operate at the rate of its slowest sink. Thus, adding many slaves slows down the rate of media flow. One solution is to use a different type of duplicator module that is not constrained to run both output streams at the rate of the slower of the two, such as the VsLossyDup module presented in chapter three.

5.6.1 Implementation

Initial Setup

The VsMultiCast application is implemented as a single Tcl script that uses the VuDP remote evaluation mechanisms to send a video stream to multiple remote sinks. Though the current implementation only allows for the remote sinks to be windows, modifying the implementation to allow for arbitrary media sinks is a natural extension.

A VsMultiCast session consists of a single "master" window and an arbitrary number of "slave" windows. The master window is the window created at startup which can control and configure the video stream and can add and delete slave windows. Slave windows only display the video stream, and can not reconfigure the video source.

Figure 5.14 illustrates the master/slave relationship. The master window is controlled by user A on host X; it maintains connections to two slave windows on hosts Y and Z. The master controls the video source, and sends copies of video stream to the slave windows on hosts Y and Z.

When started, the application script first creates the local master window, creates the video source, and creates a screen sink inside the master window. It then creates a VsNullSink module and a VsDup module. (A VsNullSink is a media sink that simply deletes the data passed to it.) A VsDup module is a duplicator used to split a single input media stream into two identical output media streams. It has one input port and two output ports; when data arrives at the input port it is copied, and one copy is sent to each output port, as seen in figure 5.15.

The output of the video source is connect to the duplicator module. One VsDup output is connected to the null sink, and one output is connected to the screen sink.

Connection State

An arbitrary number of slaves can be added to and deleted from a VsMultiCast session. In order to keep track of the various connections that exist at any one time, the application maintains one variable and three lists to provide state information - the "numrev" variable, the "allhosts" lists, the "connections" list, and the "servers" list.

The “numrev” variable contains the number of connections that have been established so far during the session. It is initially 0 when the master window has just been created, and is incremented each time a new host is added.
The “allhosts” list contains the names of all hosts currently attached to the VsMultiCast session. Hosts are added to this list when they join the VsMultiCast session and are deleted from it when they leave the session.
Each connection to a remote host has associated with it both a VsREV module and a VsTcpServer module. Hence, the “connections” list contains the names of the VsREV modules used for existing connections, and the “servers” list contains the names of VsTcpServer modules for existing connections. The “allhosts”, “connections”, and “servers” lists are maintained such that the index for a certain host in the “allhosts” list will index the VsREV and VsTcpServer module names for that connection in the “connections” and “servers” lists, respectively.

Adding New Hosts

The master window has two input boxes - one for adding new hosts and one for deleting hosts. The “Add New Host” box is linked via a callback to the “newHost” routine for adding new hosts so that the routine is called whenever the user enters a new host name into the box.
When the newHost routine is called, it adds the specified host to the multicast session. First, the routine reads the text contained in the new host widget to determine the name of the host to add and increments the “numrev” variable. It then creates a new VsREV module to handle sending data to the new host. Because the VsREV modules for all connections are maintained in the same address space, the connection number is used in the name of the VsREV module to distinguish the connections. The name of the new host is then appended to the “allhosts” list, and the name of the new VsREV module is appended to the “connections” list.
Once the new display has been created on the remote interpreter, the VsREV module is used to create a remote top-level widget and a video screen media sink linked to the new display. Then, the application creates a local VsTcpListener module listening at an unused port, and sets up a callback procedure to be executed when the listener establishes a connection. Finally, the application uses the VsREV module to create a remote VsTcpClient module, to connect its output to the input of the remote video sink, to configure it to connect to the port of the local listener it just created, and to start the remote VsTcpClient.
When the local VsTcpListener’s callback is executed, the newly created remote VsTcpClient has connected to it. Thus, an in-band connection has been established to the new host via the VsTcpServer module created by the local listener. The name of the new VsTcpServer is then appended to the “servers” list.
Finally, a copy of the media stream must be routed to the new remote display via the newly created in-band connection. To do this, a a new VsDup module is first

created. Note that, because of the way VsMultiCast is set up, there will always be some existing VsDup module with one of its outputs connected to a VsNullSink. Hence, the VsDup output that was connected to the VsNullSink is instead connected to the input of the new VsDup module. (The name of the VsDup module that is connected to the VsNullSink can be found by simply asking the null sink what its input is bound to.) One output of the new VsDup module is then connected to the VsNullSink, and one output is connected to the input of the new VsTcpServer module. Thus, the new VsTcpServer module receives a copy of the video stream to display on the screen of the new slave.
As an example, say that host X is running the master window, and is connected to slave windows on hosts Y and Z. Three copies of the video stream are needed - one for the master window on X, and one for each of the slave windows on Y and Z. The way that the video source, VsDup, VsTcpServer, and VsNullSink modules are configured is illustrated in figure 5.15.

Deleting Slaves

There are two ways of deleting slaves from a session. The first way to remove a slave from a session is to click the "dismiss" button on a slave window. Every in-band connection to a slave window has a VsTcpServer module associated with it, and each such module has a input callback that is executed when an end-of-file message is received. Because dismissing a remote slave will cause an end-of-file message to be sent, the callback is automatically performed when the "dismiss" button on a remote slave window is clicked. The VsTcpServer's end-of-file callback procedure simply calls the oldHost procedure to delete the host, supplying the name of the host to be deleted via a "-host" argument, and passing an "-eof" argument to let the oldHost procedure know that it was called because of an end-of-file condition.
The other way to delete a slave is to enter the name of the host into the "Remove Host" box in the master window. The "Remove Host" box is linked via a callback to the oldHost procedure, so that the procedure is called whenever a new host name is entered into the box.
When the oldHost procedure is run, it first checks to see if it received a "-eof" argument; if so, it means that the host to delete by clicking on a remote dismiss button, and so it can read the name of the host to delete from the "-host" argument. If there is no "-eof" argument, it means that the procedure was called because a host name was entered into the "Remove Host" box in the master window. If this is the case, the procedure reads the text in the widget to determine the name of the host to delete.
To delete a host, the procedure searches its lists of existing hosts to find the names of the VsREV and VsTcpServer modules associated with that connection, and stops the VsTcpServer module. Then, the procedure finds the name of the VsDup module whose output is connected to that VsTcpServer. Because this copy of the

media stream is no longer needed, whatever is connected to the input of this VsDup module is connected directly to the other output of the VsDup. The VsREV, VsTcpServer, and VsDup modules associated with this connection are then destroyed, their names are deleted from the "connections" and "servers" lists, and the name of the deleted host is removed from the "allhosts" lists.

Multicast and the In-Band Feedback Loop

As can be deduced from the implementation, VsMultiCast copies the video stream and sends it to all the slave windows via multiple point-to-point connections. Because one of the design decisions in developing the VuNet was to only support unicast, this copying is necessary.

VuSystem modules which handle in-band data use a feedback loop to regulate the rate of flow of media through the in-band pipeline. A video source can be configured to transmit video data no faster than a certain rate, but the actual rate at which it will send data to the sink is controlled by a feedback loop. When the source sends a new payload of video data, it must wait for the data to be accepted by the sink before it can send any more payloads down the pipeline. This feedback forces the source to produce data only as fast as the sink can accept it.

When a media stream is duplicated in a VsDup module, new input is not accepted on its input port until the last data it sent out has been accepted by both output ports. Hence, the media will flow at the highest rate acceptable to both of its outputs. When several VsDup modules are connected together (as in VsMultiCast), the media flow will travel only as fast as is acceptable to all outputs connected to any VsDup module. This means that the media stream in the VsMultiCast will travel at the rate of its slowest host.[8] The slowest host may not necessarily be a slave, though - it might be the master.

5.7 Perspective

An interesting extension of VsTalk would be to include support for handling multiple participants. This would involve integrating features of VsChat and VsMulticast into VsTalk. VsChat functionality allows for the tracking of participants in a session, and VsMulticast functionality allows for sending audio and video streams to multiple destinations.

A VuDP video conferencing tool would provide an interesting alternative to existing MBone tools. In essence, VuDP is able to provide tools that are easier to set up and to use by leveraging the simplifications available for users on machines in a single administrative domain. With VuDP, machines with a shared file system

[8] TCP is windowed, so the performance of sending data over the network used to send the data over the network is reasonable.

in the same authentication domain could use VuDP video conferencing over existing networks and routers.

```
set a 10
 :10
vs.rev revsend "set a 1"
 :
set b 5
 : 5
vs.rev revsend "set c $b"
 :
vs.rev revsend "set d {$a}"
 :
vs.rev flush
 :
vs.rev rev "set c"
 : 5
vs.rev rev "set a"
 : 1
set a
 : 10
```

Figure 5.4: Example of Remote Evaluation

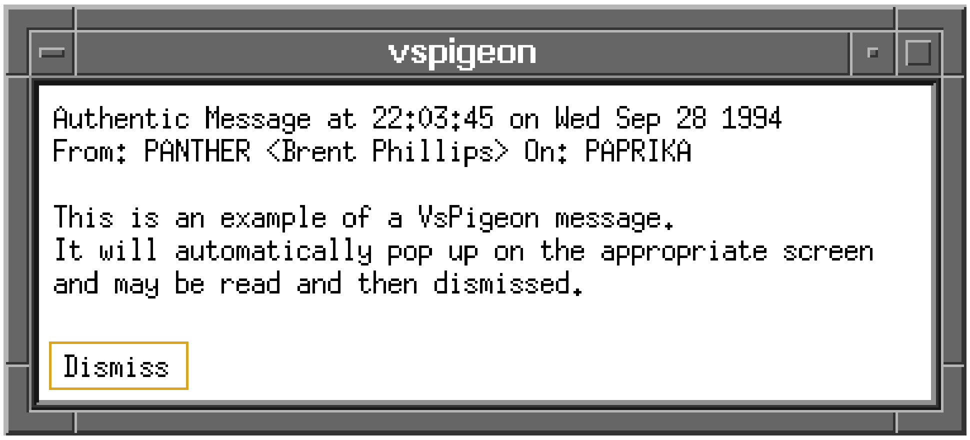

Figure 5.5: VsPigeon Message

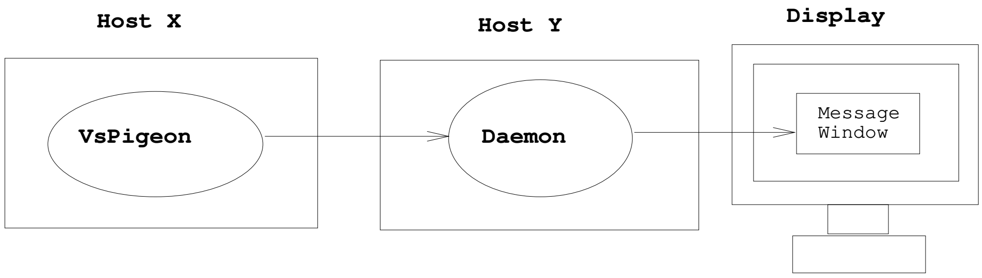

Figure 5.6: VsPigeon

Figure 5.7: VsTalk

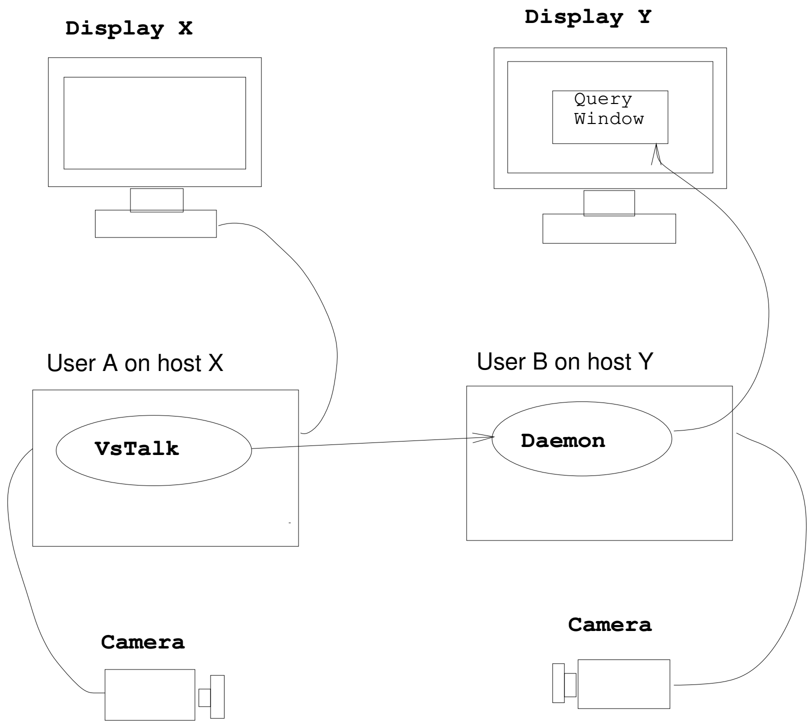

Figure 5.8: VsTalk Initiation

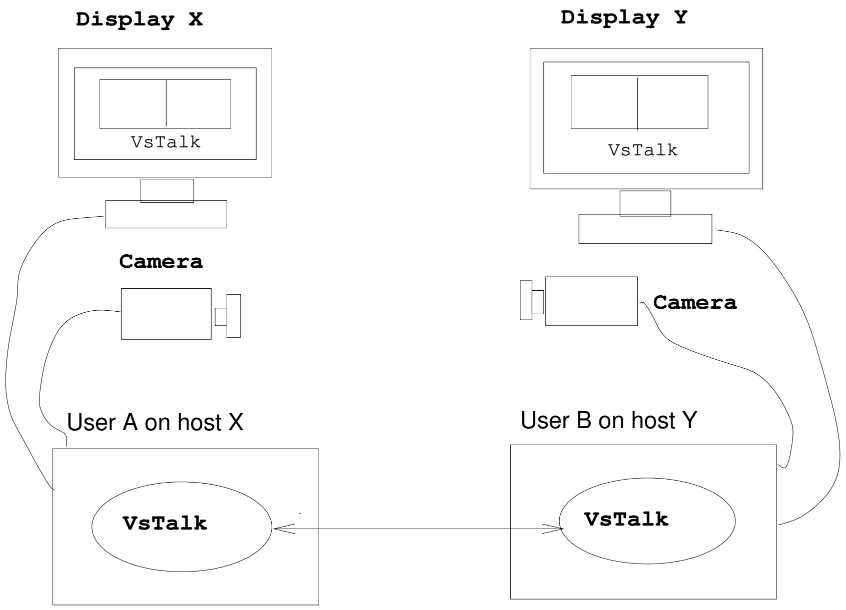

Figure 5.9: VsTalk Connection

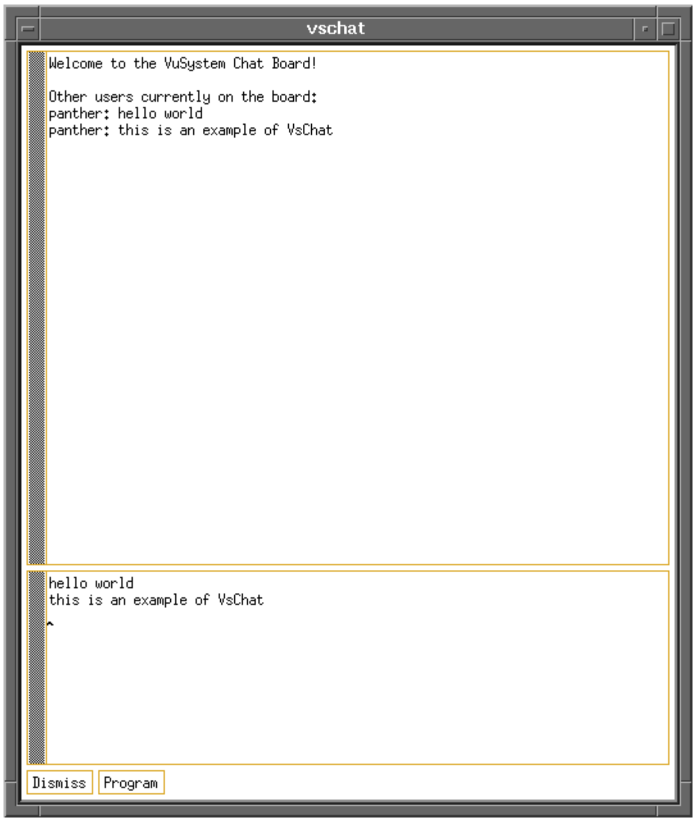

Figure 5.10: The VsChat Board

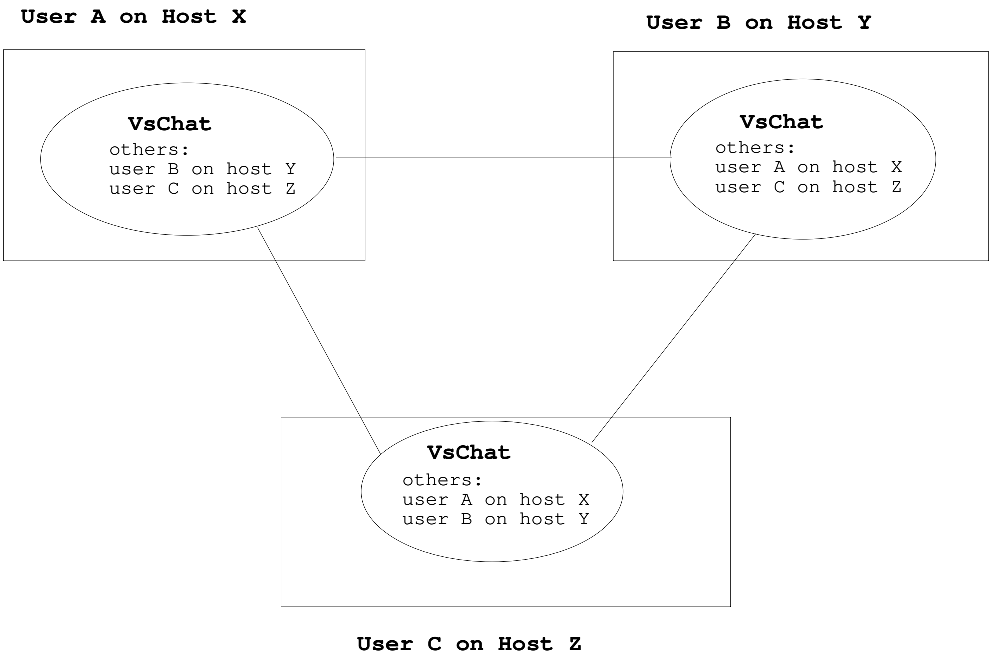

Figure 5.11: VsChat Connections

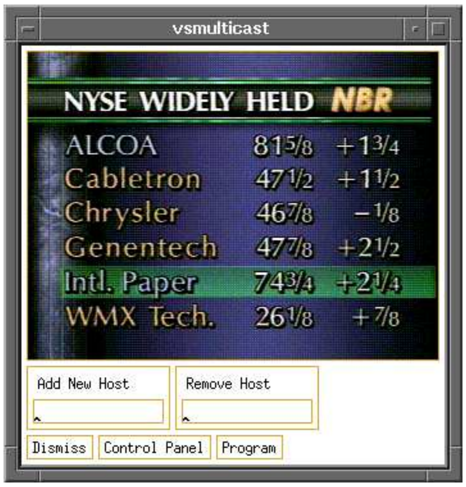

Figure 5.12: VsMultiCast Master Window

Figure 5.13: VsMultiCast Slave Window

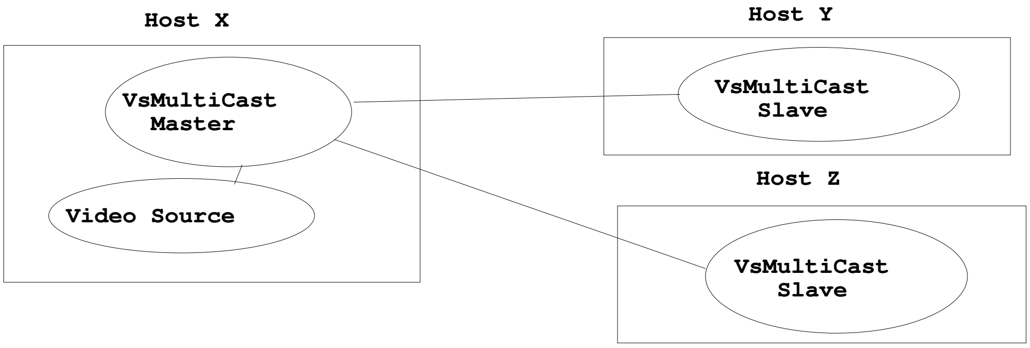

Figure 5.14: VsMultiCast Connections

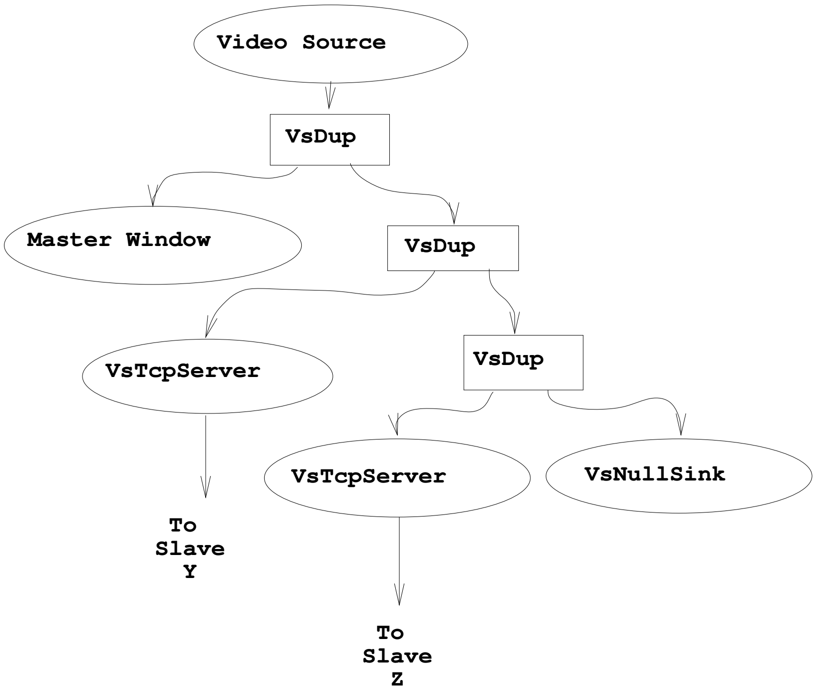

Figure 5.15: Media Duplication Connections

Chapter 6

Results

This chapter presents the results of VuDP performance testing. The most important performance difference between VuDP applications and traditional VuSystem applications is that VuDP applications must send in-band media flows over the network[1]. Different configurations of a simple VuSystem application were tested to measure the performance impact of using VuDP to send in-band media streams over the network.
To assess VuDP performance, throughput and jitter measurements were made on the video stream of the VsDemo application. VsDemo is a simple VuSystem application in which the video stream originates from a VsVideoSource module and is fed directly into a VsWindowSink module. Three different versions of VsDemo were tested:

1. Traditional VuSystem VsDemo with a local video source sending to the local display, as shown in figure 6.1. This configuration will be referred to as the "Local VsDemo".

2. VuDP VsDemo with a remote video source sending over the ethernet to the local display, as shown in figure 6.2. This configuration will be referred to as the "VuDP VsDemo".

3. Traditional VuSystem VsDemo with a local video source using X-Windows to send over the ethernet to a remote display, as shown in figure 6.3. This configuration will be referred to as the "X Windows VsDemo".

These three different versions allow the performance of VuDP remote sources to be compared against both having a local source and using X-Windows to send a video stream to a remote screen display.
All trials were run on DEC Alpha 3000 workstations running the VuSystem on top of OSF 1.3 and connected by a ten megabit per second ethernet LAN. Video

[1]Out-of-band throughput performance is not critical (hence the name "out-of-band") because out-of-band communication is generally very low bandwidth.

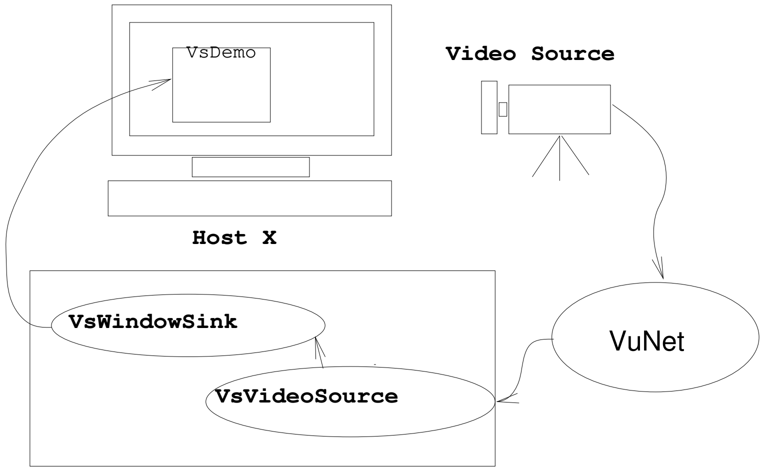

Figure 6.1: Local VsDemo

Configuration	Local	VuDP	X
640x480, color	3.75	3.33	3.33
640x480, mono	9.5	3.33	3.33
320x240, color	10	10	10
320x240, mono	15	13	10

Table 6.1: Throughput in Frames/Sec

sources were live television feeds captured and digitized by vidboards [2]. For the VuDP VsDemo and the X Windows VsDemo configurations, the remote machine was not otherwise loaded.

6.1 Throughput Performance

In order to better isolate the different components of performance, throughput was measured for full scale (640x480) eight-bit color video, full scale eight-bit monochrome video, half scale (320x240) eight-bit color video, and half scale eight-bit monochrome video. Measurements were made by observing the rate meters in the control panels for the VsVideoSource and VsWindowSink modules. The average throughput in frames per second for each configuration is shown in table 6.1. The average throughput in kilobytes per second for each configuration is shown in table 6.2.

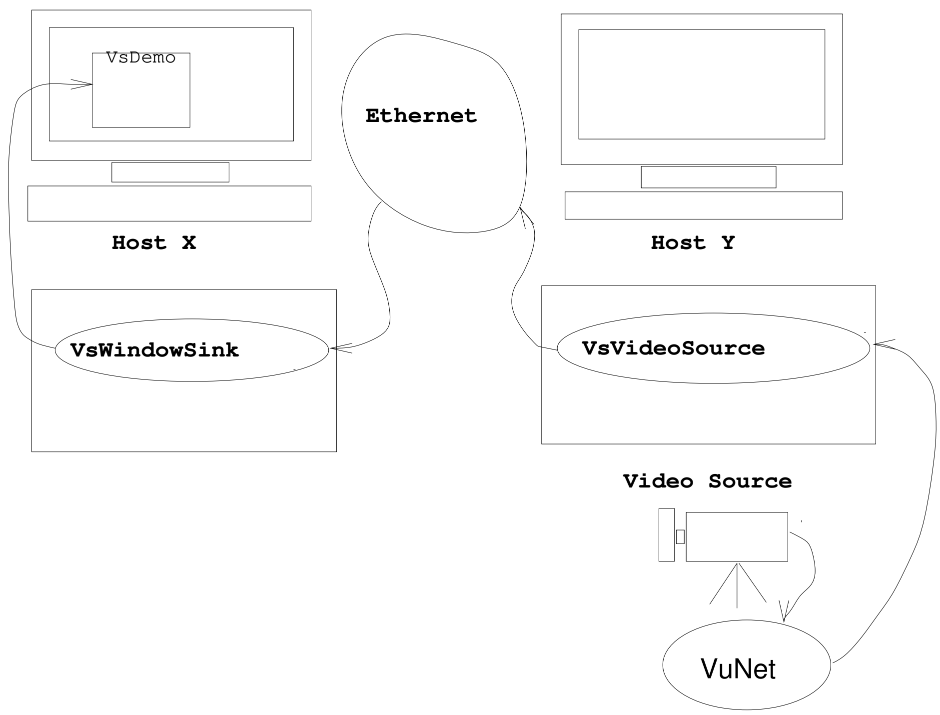

Figure 6.2: VuDP VsDemo

6.2 Jitter Performance Testing

Jitter is an important component of performance for media applications because jitter (especially in live video streams) is visible to human users. It is especially important to measure jitter in assessing VuDP performance, since it is expected that the main performance gain from using a lossy VuDP module data protocol would be reduced jitter.

When a video source is configured with the desired frame rate, a timeout is set up to expire when it is time to process a new frame.[2] Jitter is then measured by recording the delay in milliseconds between the timeout and when the corresponding video frame is passed to (and accepted by) the window sink module. Note that this measure of delay will always be below the delay between the timeout and the point when the video frame is actually displayed on the screen. Thus, these delay measurements do not directly include any delay between the

[2]If the video source already has a frame in its buffer that has not been sent downstream yet, the timeout is ignored. This serves to match the constant frame rate produced by the video source to the dynamically varying rate of the video sink. In essence, the video source's frame rate provides an upper bound on the end-to-end frame rate.

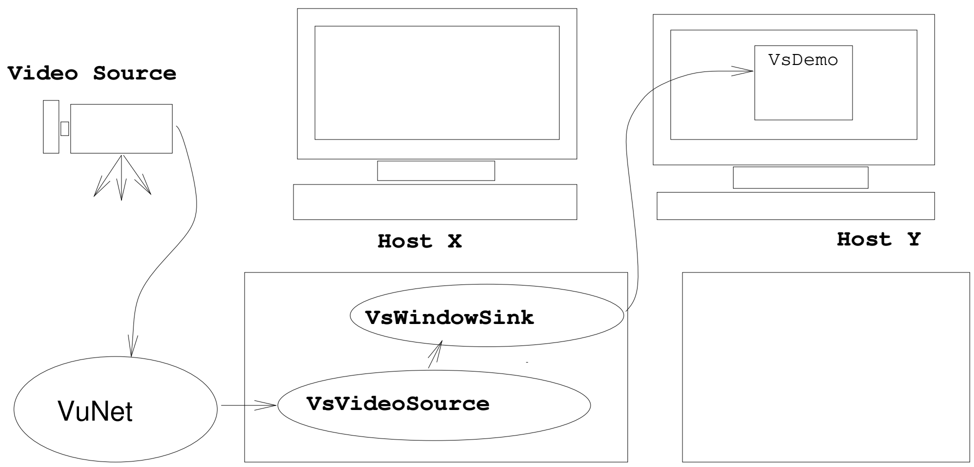

Figure 6.3: X-Windows VsDemo

Configuration	Local	VuDP	X
640x480, color	1,125	1000	1000
640x480, mono	2,850	1000	1000
320x240, color	750	750	750
320x240, mono	1,125	975	750

Table 6.2: Throughput in Kbytes/Sec

time when VsWindowSink module accepts the new video frame and the time when X Windows actually displays the video frame on the screen.[3] For the local VsDemo and the VuDP VsDemo, this delay should be small, since the X display that the video stream is being sent to is on the screen physically attached to the local host. This is illustrated in figures 6.1 and 6.2 where the VsWindowSink module writes to the screen of the local host. For the X VsDemo, though, this extra delay also includes the time it takes for X Windows to send the video stream over the network to the host physically attached to the screen that the display is on. This is illustrated in figure 6.3, where the VsWindowSink module writes to a display on a remote screen. The way that this difference affects the meaning of the jitter measurements is discussed in more detail in the next section.
Because jitter is likely to be sensitive to many effects other than the local or remotes nature of the video source and window sink, jitter measurements were

[3]This delay is indirectly measured because the VsWindowSink module will not accept a new video frame until the last frame has been displayed.

performed in ten separate trials for each version of VsDemo. For each trial, VsDemo was configured to run at half-scale (320x240) in eight-bit color for approximately five minutes to generate data for about 10,000 video frames. For each frame, the delay was recorded as the time in milliseconds that elapsed between the timeout and the point at which the video frame was accepted for display by the VsWindowSink module.
Important jitter statistics are presented in the tables 6.3, 6.4, and 6.5 for the local, VuDP, and X-Windows configurations, respectively. Each table presents the median and range of jitter measurements over the ten trials.

Delay in ms	Range	Median
Below 1 ms	64.8% to 92.9%	91.0%
Below 33 ms	88.5% to 99.8%	99.5%
Below 100 ms	99.5% to 99.9%	99.8%
Below 200 ms	99.6% to 99.9%	99.9%
Below 500 ms	99.8% to 100%	99.9%
Below 1 sec	99.9% to 100%	100%

Table 6.3: Performance Statistics for Local VsDemo

Delay in ms	Range	Median
Below 1 ms	57.0% to 88.0%	87.0%
Below 33 ms	79.7% to 99.4%	98.4%
Below 100 ms	98.4% to 99.8%	99.0%
Below 200 ms	98.9% to 99.9%	99.5%
Below 500 ms	99.6% to 100%	99.8%
Below 1 sec	99.9% to 100%	100%

Table 6.4: Performance Statistics for VuDP VsDemo

These statistics demonstrate interesting characteristics of in-band flows. For both local and remote sources, the majority of the delays between timeout and display are less than one millisecond, implying that the median delay is less than one millisecond. A large majority of delays are less than 33 ms, and very few are greater than 100 ms.
Data for two representative trials for each of the three configurations are presented in the graphs in figures 6.4, 6.5, 6.6, 6.7, 6.8, and 6.9. The delay in milliseconds is plotted along the X-axis, and the percentage of video frames incurring that delay is plotted on the Y-axis. Note that the Y-axis is a base ten logarithmic scale. All

Delay in ms	Range	Median
Below 1 ms	36.1% to 64.5%	55.0%
Below 33 ms	73.5% to 85.8%	80.6%
Below 100 ms	93.6% to 99.9%	98.6%
Below 200 ms	96.3% to 100%	99.5%
Below 500 ms	98.9% to 100%	99.9%
Below 1 sec	99.5% to 100%	100%

Table 6.5: Performance Statistics for X-Windows VsDemo

delays are rounded down, so that the zero milliseconds marker corresponds to all delays of less than one millisecond, the one millisecond marker corresponds to all delays between one and two milliseconds, etc.

6.3 Analysis of Throughput Measurements

The throughput numbers demonstrate two throughput bottlenecks for different configurations of VsDemo.

The first bottleneck is sending a media stream over the ethernet. Note that there is a consistent upper bound of approximately one megabyte per second throughput for both the VuDp VsDemo and X VsDemo. This upper bound is most likely the result of sending the media stream over the network. This is quite reasonable, considering that one megabyte per second throughput corresponds to 80% utilization of the underlying ten megabit per second network[4]. Further, the VuDP VsDemo and X VsDemo have nearly identical throughputs for all cases (except for 320x240 monochrome, where VuDP does slightly better), implying that network communication is the bottleneck. The difference for 320x240 monochrome is likely to result from the overhead X Windows incurs, meaning that VuDP is slightly more efficient than X in transferring media flows across the network.

The second bottleneck is the processing required for dithered color video. For the local VsDemo, note that the throughput for 640x480 in eight-bit monochrome is much better than the throughput for 640x480 in eight-bit color. This indicates that the color processing is the performance bottleneck. In other words, the throughput is computation limited and not communication limited. This observation is backed up by comparing the local VsDemo throughput to the VuDP VsDemo and X VsDemo. For color video, the local throughput is very close (for 640x480) or identical (for 320x240) to the throughput for the VuDP VsDemo and X VsDemo. But, for monochrome video, the local throughput is significantly better than the

[4]This 80% utilization does not take into account TCP/IP, ethernet, and VuSystem payload overheads, so the actual utilization is even higher.

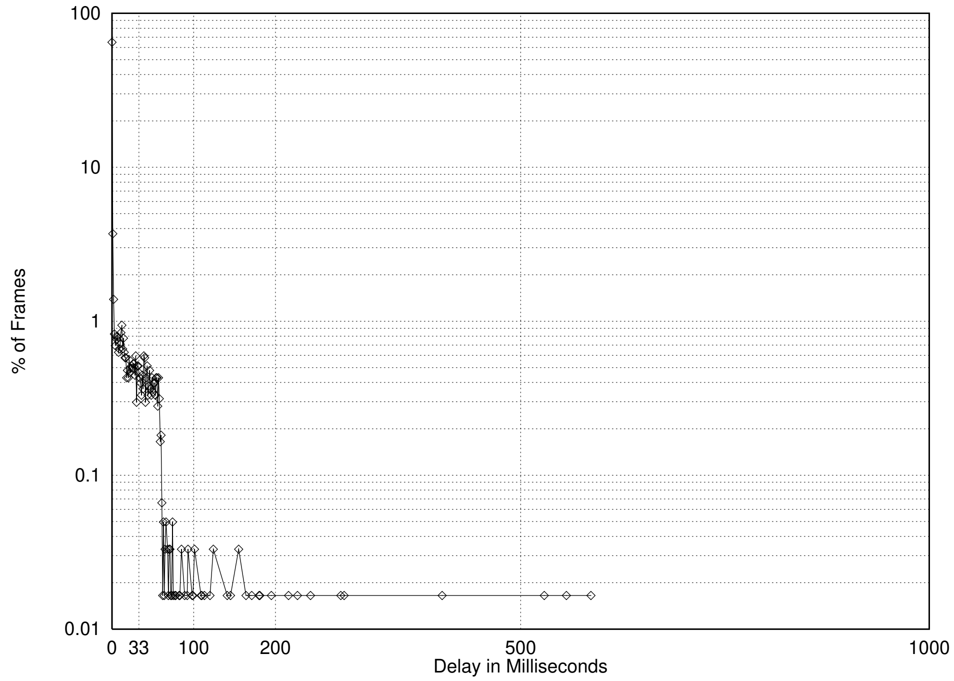

Figure 6.4: Local VsDemo, Trial 1

throughput for VuDP or X VsDemo. Thus, network communication limits the throughput for monochrome video, while the computation required to handle the color dithering is the throughput bottleneck for eight-bit color video.

6.4 Analysis of Jitter Measurements

The jitter measurements are quite interesting. Most delays are less than one millisecond, whether or not the video is transferred over the network. This indicates that latency incurred by transferring data over an unloaded network is not an issue.

Note that the range of results across trials for a given configuration are much greater than the variance in the measurements across configurations. This indicates that factors other than the local or remote nature of the video source are very important in determining jitter[5].

As a result of the way jitter is measured, it is difficult to directly compare the X version to the local and VuDP versions. As mentioned above, jitter is measured by

[5]Other factors might include but are not limited to network loading and other processes running on the machines.

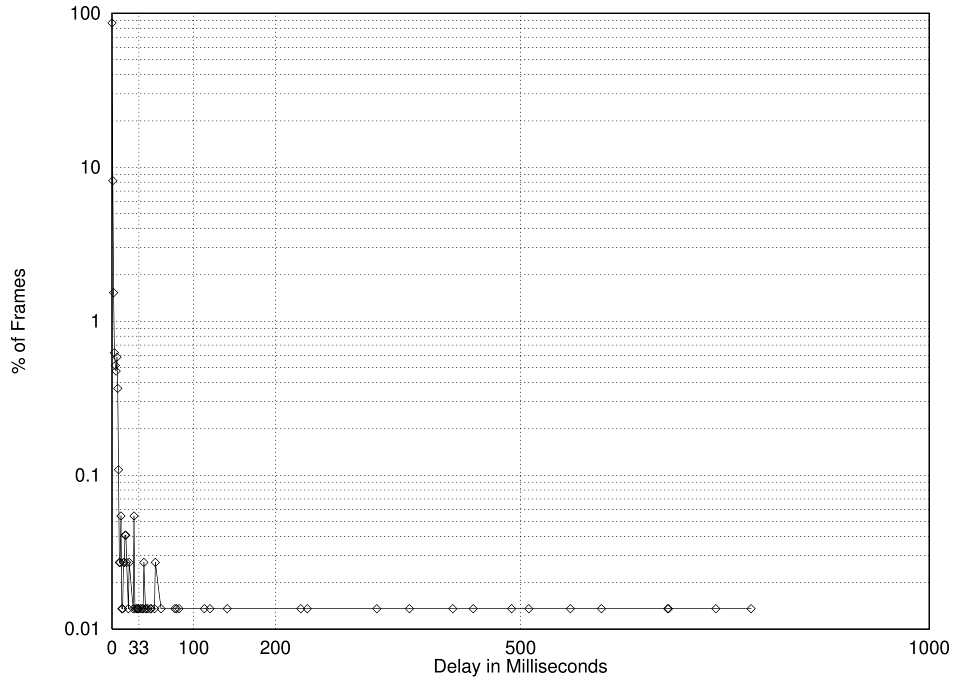

Figure 6.5: Local VsDemo, Trial 2

recording the delay between when the timeout goes off for a certain video frame and when that video frame is accepted by the VsWindowSink for display to the screen. Hence, the time that it takes for the frame to be transferred to the screen is not directly included in the calculation. For the X version of VsDemo, the video frame must be transferred across the network prior to display, and so this delay is only indirectly included in the jitter measurements because the back pressure of the module data protocol will not allow the VsWindowSink to accept a new payload until the last payload has been displayed.

Comparing the jitter of the local VsDemo to the VuDP VsDemo, it appears that the local VsDemo has slightly less jitter on average. The plots for the local VsDemo trials have slightly higher peaks for low delays and thinner tails than the VuDP VsDemo plots, inferring that sending the media stream over the network does increase jitter. However, the increase is small, and it must be remembered that the differences across trials swamp the differences across configurations.

Finally, note that while there is great variation in the jitter there is also a very high percentage of frames with jitter below 33ms. One of the most important characteristics of the VuSystem is that it runs in perceptual time. "Perceptual time" means that the VuSystem runs in virtual time without the real time constraints and overhead because it is assumed that the hardware and software are

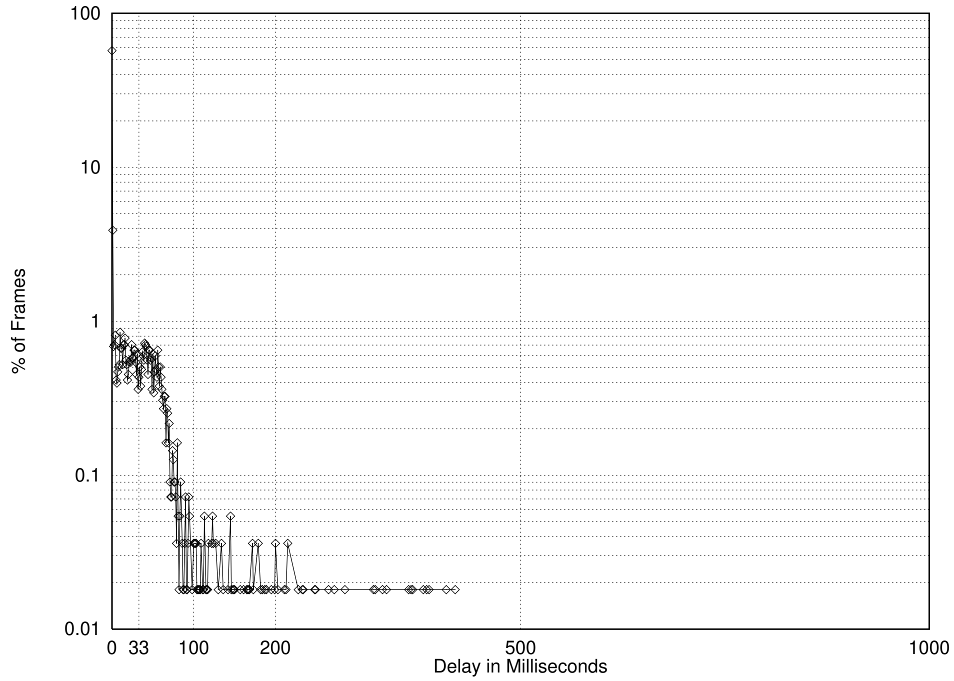

Figure 6.6: VuDP VsDemo, Trial 1

fast enough to appear to the user to be running in real time. Film to video conversions routinely used in broadcast television introduce 15 ms jitter per frame and are considered to be acceptable quality, so the results presented here support the perceptual time assumptions.

6.5 A Lossy VuDP Protocol?

The performance discrepancies between the local VsDemo and the VuDP VsDemo place an upper-bound on any gains from using a lossy protocol for media transport over the network. The performance difference is small, but not insignificant. However, some increase in jitter is an unavoidable result of network communication, no matter what protocol is used. Further, in informal experiments involving viewing VsDemo using both local and remote video sources, there does not seem to be an obvious visible difference in performance.

It is important to remember that the difference in jitter across trials is much greater than the difference in jitter between the local VsDemo and the VuDP VsDemo. Still, it appears that there is some room for improvement by changing VuDP to use a lossy protocol when transferring media streams over the network.

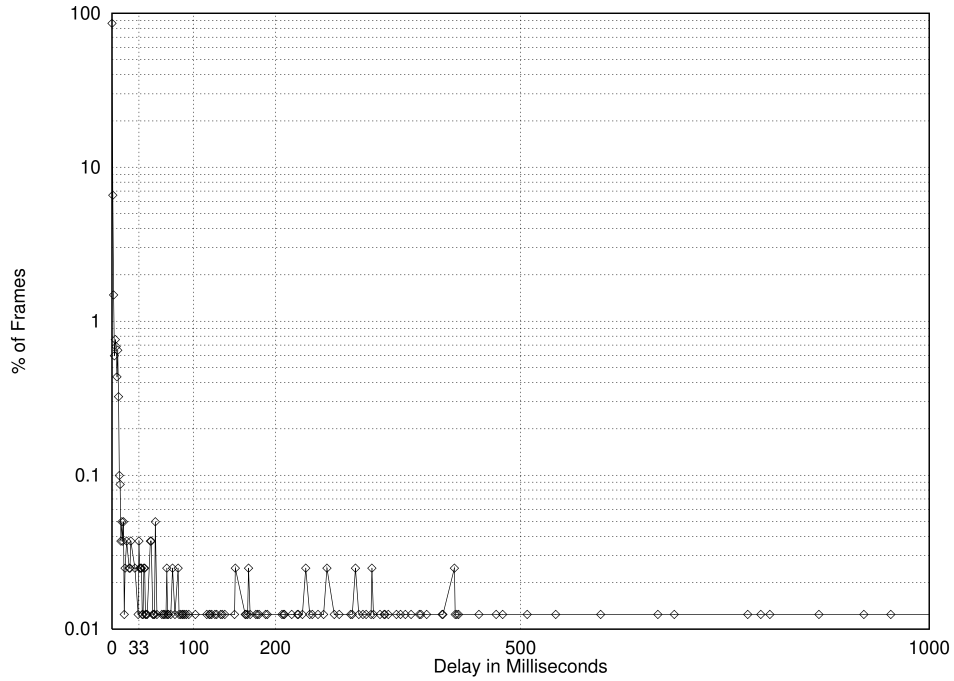

Figure 6.7: VuDP VsDemo, Trial 2

On the other hand, two important advantages of the current VuDP protocol are that it is simple to use and understand and elegantly extends the VuSystem module data protocol. A new lossy protocol would be most likely improve jitter a little bit and maybe improve throughput a little bit. But a new lossy protocol would also incur the disadvantage of complicating VuDP while still needing to deal with end-to-end flow control, network congestion flow control, and reordering and lost packets without retransmission. If the decrease in jitter is deemed worth the effort put into developing a new protocol and complexity added to VuDP, then it makes sense to use a lossy protocol. Otherwise, the current VuDP module data protocol works well and is sufficient.

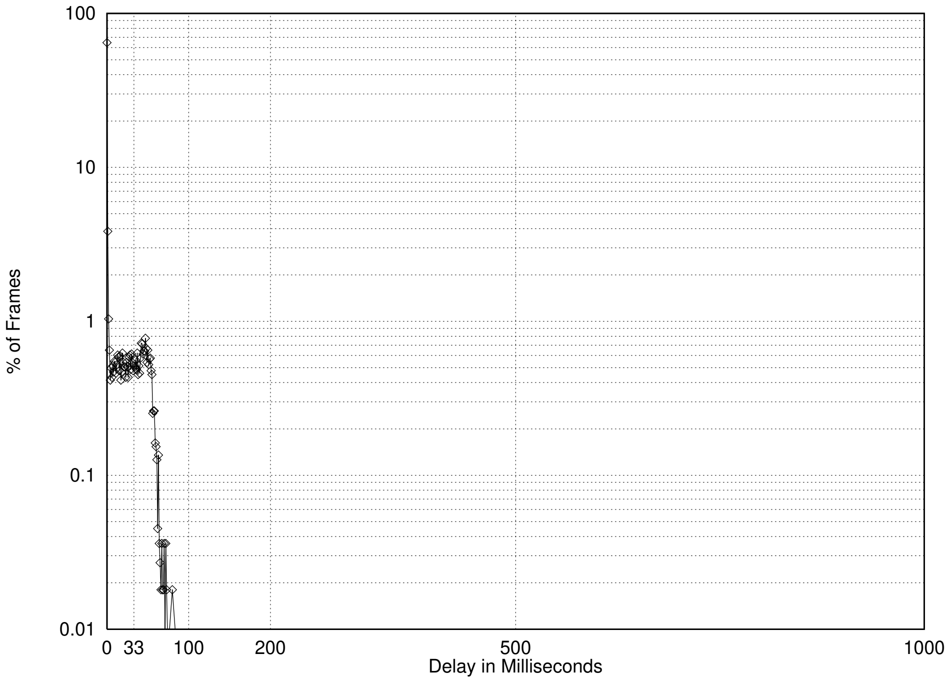

Figure 6.8: X Windows VsDemo, Trial 1

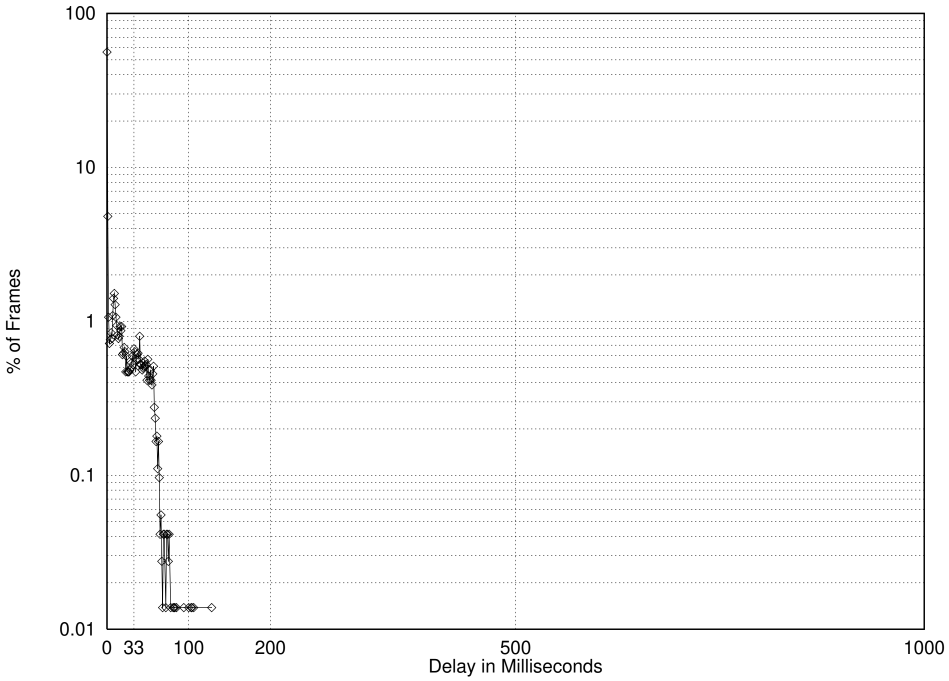

Figure 6.9: X Windows VsDemo, Trial 2

Chapter 7

Conclusion

The design and implementation of VuDP has shown the feasibility of a distributed programming system for media applications. Both the basic VuDP facilities and the more advance remote evaluation and VuDP daemon mechanisms have proven to be useful tools for developing distributed VuSystem applications. This chapter concludes this thesis, explains lessons learned from VuDP, and provides ideas for future work.

7.1 VuDP Summary

The VuDP implementation has been largely successful in meeting its design goals. Using the VuDP extensions, it is a simple matter to build distributed VuSystem applications to take advantage of all the resources of the machines attached to a local area network. VuDP has also been smoothly integrated with the VuSystem runtime programming and dynamic configuration capabilities. Further, the performance of VuDP applications is also quite good, since throughput is nearly as good as for local applications and jitter is within reasonable bounds. In the future, VuDP's performance will continue to improve as computers become faster.

The VuDP remote sourcing capability has been found to be particularly convenient. Often only one video source is attached to a workstation, and if it is in use (for example, if it is being used to perform a daily recording) then that workstation can not be used to run or develop applications that require a video source. The ability to use remote sources and sinks greatly enhances the flexibility of running and developing applications that require the use of scarce resources.

7.2 Important Lessons

Three important lessons have been learned from the design and implementation of VuDP. First, the interaction of *rsh* and a network file system simplifies distributed programming. Second, remote evaluation is a powerful tool for building both

distributed applications and new distributed programming services. Finally, debugging distributed programs is difficult and awkward.
The interaction between *rsh* and the network file system was found to be extremely useful for implementing VuDP. Most of the VuDP functionality is based on the ability to easily and simply run a script on a remote machine. The network file system allows any application script on a disk attached to any machine to be run from any other machine. *rsh* allows an application to simply and easily start a script on a remote machine and automatically takes care of all authentication. Because authentication is performed by *rsh*, the network file system's permissions can be used to control remote file access so that users can remotely access only the appropriate files.
Originally, VuDP was designed to use the VuDP daemon for all remote programming services. However, when this was found to be too constraining, the general remote evaluation mechanism was introduced. Once the remote evaluation was developed, it was found that some of the VuDP services that had previously used the daemon - such as the remote sourcing capability - were simpler and more powerful when implemented directly on top of *rsh*.
Another important point that made itself abundantly clear during the development of VuDP is that it is very difficult to debug distributed programs. Most debugging tools are only designed to work with an application that works within a single address space, and so are not very helpful for debugging applications that span several address spaces on several machines. Further, even the most rudimentary debugging tools - such as printing diagnostic output to the screen - are more difficult to use with distributed programs. For example, standard output from applications started with *rsh* is not normally displayed on the screen. Finally, developing and debugging distributed applications is difficult because it is necessary to have access to multiple functional workstations and a functional network. If there are not enough available workstations, or if there are problems with the underlying network, it becomes difficult or impossible to write distributed applications. At any given time, the probability of having a single functional workstation is much greater than the probability of having several functional workstations and a functional network.

7.3 Future Work

There are several small projects which would be natural extensions to VuDP. The first is to integrate media into the VsChat application, so that users may share audio and video images as well as text. A second is to redesign the control panel mechanism so that the control panels for remote devices are completely integrated into the local control panel. A third is to redesign the VsEntity so that any VuSystem application entity can be placed remotely at any host and dynamically moved from host to host at run time, much as can be done for remote filters with

the current VuDP.
The design and implementation of a lossy protocol for video transport is another logical extension to the VuSystem. As discussed previously, such a protocol has the potential to provide less jitter and possibly slightly better throughput than TCP.
The traditional VuSystem supports only single-threaded applications. This was a major concern during the design of VuDP, since it is not possible to create separate threads to handle high-latency network communication of out-of-band data. Thus, another interesting extension that VuDP could take advantage of would be the addition of a multi-threading facility to the VuSystem.
Finally, an interesting project would be to modify the VuSystem so that the in-band portion of all modules can be remotely placed on any host, but so that all out-of-band code for any application resides in a single local address space. This takes advantage of the way that applications are split into in-band and out-of-band sections, and would both simplify and increase the power of distributed VuSystem applications. This would be a major undertaking, though, since doing this would require redesigning and reimplementing a substantial portion of the VuSystem infrastructure.

Bibliography

[1] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. Mach: A new foundation for Unix development. In *Summer Conference Proceedings*. USENIX Association, 1986.

[2] Joel F. Adam. The vidboard: A video capture and processing peripheral for a distributed multimedia system. In *Proceedings of the ACM Multimedia Conference*. ACM, Aug 1993.

[3] Joel F. Adam, Henry H. Houh, and David L. Tennenhouse. A network architecture for distributed multimedia systems. In *Proceedings of the International Conference on Multimedia Computing and Systems*, pages 76–86, May 1994.

[4] Joshua Auerbach, Arthur P. Goldberg, German S. Goldszmidt, Ajei Gopal, Mark T. Kennedy, and James Russel. Concert/C manual: A programmer's guide to a language for distributed computing. Technical Report IBM RC 19232, IBM Research Center, Yorktown Heights, NY, Oct 1993.

[5] Mario R. Barbacci and Jeanette M. Wing. A language for distributed applications. In *Proceedings of the International Conference on Computer Languages*, pages 59–68. IEEE, May 1990.

[6] Kenneth P. Birman. Isis: A system for fault-tolerant distributed computing. Technical Report TR-86-744, Cornell University Dept. of Computer Science, July 1987.

[7] S.M. Clamen, L.D. Leibergood, S.M. Nettles, and J.M. Wing. Reliable distributed computing with Avalon/Common Lisp. Technical Report CMU-CS-89-186, Carnegie-Mellow Univ. Computer Science Dept., Sep 1989.

[8] Douglas E. Comer and David L. Stevens. *Internetworking with TCP/IP*, volume III: Client-Server Programming and Applications, BSD socket version. Prentice Hall, 1993.

[9] Arthur Goldberg, Robert Strom, and Shaula Yemini. Hermes: A high-level process-model language for distributed computing. Technical Report IBM RC 17707, IBM Research Center, Yorktown Heights, NY, Feb 1992.

[10] Brent Hailpern and Gail E. Kaiser. An architecture for dynamic reconfiguration in a distributed object-based programming language. Technical Report IBM RC 18269, IBM Research Center, Yorktown Heights, NY, Aug 1992.

[11] A. Hopper. Pandora - an experimental system for multimedia applications. *ACM Operating Systems Review*, April 1990.

[12] Clifford Dale Krumvieda. *Distributed ML: Abstractions for Efficient and Fault-Tolerant Programming.* PhD thesis, Cornell University, Aug 1993.

[13] C. J. Lindblad, D. Wetherall, and D. L. Tennenhouse. The VuSystem: A programming system for visual processing of digital video. In *Proceedings of ACM Multimedia*, October 1994.

[14] Christopher J. Lindblad. *A System for the Dynamic Manipulation of Temporally Sensitive Date.* PhD thesis, MIT, Aug 1994. MIT/LCS/TR-637.

[15] Barbara Liskov. Distributed programming in Argus. *Communications of the ACM*, 31(3):300–312, March 1988.

[16] Michael R. Macedonia and Donald P. Brutzman. Mbone provides audio and video across the internet. *IEEE Computer*, 27(4):30–36, April 1994.

[17] John Outerhout. TCL: An embeddable command language. *USENIX*, 1990.

[18] F.N. Parr and R.E. Strom. Nil: A high level language for distributed systems programming. Technical Report IBM RC 9750, IBM Research Center, Yorktown Heights, NY, Dec 1982.

[19] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrman, C. Kaiser, S. Langlois, P. Leonard, and W. Neuhauser. Chorus distributed operating systems. *Computing Systems*, 1(4):305–370, 1988.

[20] Stuart Sechrest. *A Client-Server Shell Architecture for Distributed Programming.* PhD thesis, University of California, Berkeley, Aug 1993.

[21] James W. Stamos. *Remote Evaluation.* PhD thesis, MIT, Jan 1986.

[22] W. Richard Stevens. *TCP/IP Illustrated*, volume I. Addison-Wesley, 1994.

[23] Mark Sullivan and David Anderson. Marionette: A system for parallel distributed programming using a master/slave model. Technical Report UCB/CSD-88/460, University of California, Berkeley, Computer Science Division, Nov 1988.

[24] D. L. Tennenhouse, J. Adam, D. Carver, H. Houh, M. Ismert, C. Lindblad, W. Stasior, D. Weatherall, D. Bacher, , and T. Chang. A software-oriented approach to the design of media processing environments. In *Proceedings of the International Conference on Multimedia Computing and Systems*. IEEE, May 1994.

[25] David J. Wetherall. An interactive programming system for media computation. Master's thesis, MIT, Aug 1994. MIT/LCS/TR-640.

[26] Jeanette M. Wing, Maurice Herlihy, Stewart Clamen, David Detlefs, Karen Kietzke, Richard Lerner, and Su-Tuen Ling. The Avalon/C++ programming language. Technical Report CMU-CS-88-209, Carnegie-Mellow Univ. Computer Science Dept., Dec 1988.

[27] Stuart Wray, Tim Glauert, and Andy Hopper. The medusa application environment. In *Proceedings of the International Conference on Multimedia Computing and Systems*, pages 265–273. IEEE, May 1994.