

DLoVe

A specification paradigm for designing distributed VR applications for single or multiple users

***A dissertation
submitted by***

Leonidas Deligiannidis

***In partial fulfillment of the requirements
for the degree of***

Doctor of Philosophy

in

Computer Science

TUFTS UNIVERSITY

February 2000

Adviser:

Dr. Robert J. K. Jacob

Abstract

Most of today's Graphical User Interfaces (GUI) and toolkits are based on serial, discrete, token based, paradigms that seem to acceptably implement traditional WIMP (Window, Icon, Menu, Pointer) interfaces. These tools however, are not suited for "next generation" interaction techniques such as Virtual Reality (VR). These interaction techniques rely upon asynchronous, parallel, and continuous user/computer interaction. This work proposes a specification paradigm, DLoVe, which provides a framework of techniques and abstractions that directly addresses these issues. In addition, DLoVe also provides a mechanism for executing programs designed for a single machine to be executed in a distributed environment, where updates on DLoVe variables are processed in parallel. DLoVe's framework also provides the abstractions for writing multi-user VR programs or transforming existing single-user programs into multi-user ones. Moreover, DLoVe addresses issues of performance and maintainability, providing mechanisms, drivers, and utilities that allow run time tuning and network management to be specified in a simple manner.

This thesis describes in detail the proposed system, evaluates it, and provides examples and analysis of several applications developed within DLoVe's paradigm, and provides a guide on how applications can be implemented in this framework.

Acknowledgments

I wish to take this opportunity to express my sincere thanks to the people who made this study possible.

In particular, I would like to express the deepest gratitude to my advisor, Prof. Robert J. K. Jacob, for his consistent encouragement, support, inspiration, and guidance throughout my studies for my Masters and Ph.D. It is my privilege to be his student.

I want to thank Prof. Alva L. Couch for dedicating so much time to me to help me analyze and measure DLoVe's performance to successfully complete my dissertation and defense. I thank him for the many hours of discussion and for his continuing effort to cultivate my ability to write clearly.

Special thanks to Prof. Karen P. Lentz for allowing me to use all her SGI machines to execute my distributed programs, and for her inspiration and ideas for DLoVe's distributed version during my Masters and Ph.D.

I would like to thank Prof. Harriet Fell for always believing in me and challenging me, encouraging me, and supporting me during my undergraduate studies. It is my privilege to be her student and I really enjoyed attending her numerous classes.

A special debt of gratitude is owed to my family back in Greece as well as my family here in Boston because without them I would not be where I am today. My greatest debt is owed to my lovely wife, Christina, for her abundant love, for her never-

ending faith in me, and for her understanding (my long absence from home) during the preparation and completion of this dissertation.

This work was supported by National Science Foundation Grant IRI-9625573, Office of Naval Research Grant N00014-95-1-1099, and Naval Research Laboratory Grant N00014-95-1-G014. We gratefully acknowledge their support.

Table of Contents

CHAPTER 1: INTRODUCTION	2
OVERVIEW.....	2
EASE OF USE.....	5
OUTLINE OF THE DISSERTATION.....	6
CHAPTER 2: RELATED WORK - VIRTUAL REALITY	9
INTRODUCTION	9
WHY USE VIRTUAL REALITY	10
LIMITATIONS IN VR.....	11
PHYSIOLOGICAL DANGERS OF VR.....	14
CHAPTER 3: RELATED WORK - CONSTRAINTS AND CONSTRAINTS IN UIMSS	
.....	16
INTRODUCTION	16
EVOLUTION OF CONSTRAINTS.....	18
<i>One-way and Multi-way Constraints</i>	18
<i>Lazy and Eager Constraint Solving</i>	20
CONSTRAINT SYSTEMS.....	21
ALGORITHMS IN CONSTRAINT SOLVING.....	26
<i>Data-Driven</i>	27
<i>Demand-Driven</i>	28
<i>The nullification/re-evaluation scheme</i>	28
DESCRIPTION OF ALGORITHMS FOR CONSTRAINT SOLVING.....	29
COMPARISON OF CONSTRAINTS	31

CONTINUOUS AND DISCRETE TIME	34
CHAPTER 4: RELATED WORK - PARALLEL AND DISTRIBUTED SYSTEMS ...	36
INTRODUCTION	36
FLYNN'S TAXONOMY	38
PERFORMANCE EVALUATION	39
PARALLEL SOFTWARE.....	42
DLOVE AND OTHER DISTRIBUTED SYSTEMS.....	45
CHAPTER 5: RELATED WORK - MULTI-USER UIMS/CSCW SOFTWARE FOR	
VIRTUAL REALITY	47
INTRODUCTION	47
SYSTEMS THAT USE DEAD RECKONING	48
UPDATE FILTERING AND AREA OF INTEREST (AOI)	52
KEY FRAME ANIMATION.....	53
OTHER USER INTERFACES	54
CHAPTER 6: BASICS OF DLOVE	60
INTRODUCTION	60
<i>Continuous Time</i>	61
<i>Discrete Time</i>	65
<i>Communication Between Continuous and Discrete Time</i>	69
DATA STRUCTURES	69
INTERNAL FLAGS OF THE SOLVER	73
OPERATIONS ON LINKS.....	75
OPERATIONS ON VARIABLES.....	76
CONSTRAINT SOLVER.....	77
<i>How it Works</i>	78

CHAPTER 7: DISTRIBUTED/PARALLEL DLOVE	84
INTRODUCTION	84
BASIC STRUCTURE OF THE 'MAIN()' FUNCTION	85
HIGH LEVEL SYSTEM ARCHITECTURE.....	87
COMMUNICATION PROTOCOL.....	91
PARTITION ALGORITHMS.....	93
<i>Partition (2 phases) (Partition Queries)</i>	94
<i>Optimization (Workers) (MyMainVars)</i>	99
PARALLEL COMPUTATION ON A DISTRIBUTED GRAPH	100
TWO FLAVORS OF CONSTRAINT SOLVERS.....	102
<i>Constraint Solver on the Coordinator</i>	103
<i>Constraint Solver on the Worker(s)</i>	104
CONFIGURATION FILE	104
CHAPTER 8: MULTI-USER DLOVE	105
INTRODUCTION	105
MULTIPLE COORDINATORS.....	106
ISSUES – TRANSFORMING TO MULTI-USER.....	108
<i>Backward Notification of Events</i>	111
<i>The Need for Local Evaluation</i>	112
<i>Time is a special Variable</i>	113
CHAPTER 9: MESSAGING (DLOVE PROTOCOL).....	115
INTRODUCTION	115
MACHINE CONNECTIONS IN DISTRIBUTED MODE.....	116
MACHINE CONNECTIONS (MULTI-USER)	118
FLAVORS OF MESSAGE-PASSING IN DLOVE.....	124

<i>Single messaging</i>	125
<i>Multi-messaging - Block for replies</i>	127
<i>Multi-messaging - Non-Block</i>	128
CONFIGURATION FILE	130
CHAPTER 10: THE ‘ARMS’ APPLICATION.....	132
OVERVIEW.....	132
OBJECTS IN ‘ARMS’ - SINGLE USER.....	134
‘ARMS’ IN DISTRIBUTED MODE	139
‘ARMS’ FOR A MULTI-USER ENVIRONMENT	142
<i>The ‘switch’ mechanism’</i>	146
CHAPTER 11: AN EYE TRACKING APPLICATION	149
OVERVIEW.....	149
HAND MOVEMENT FOR OBJECT MANIPULATION.....	150
DESIGNING THE APPLICATION.....	151
CHAPTER 12: THE DLOVE VIRTUAL PARK.....	156
INTRODUCTION.....	156
THE ACTORS.....	157
<i>Sliders and Orbiting objects</i>	160
COLLISION PREVENTION	164
<i>Implementation Issues</i>	165
DESIGNING THE VIRTUAL PARK.....	166
<i>The Virtual Ball</i>	167
<i>The Humanoids</i>	167
<i>Partition Issues</i>	169
<i>Collision Prevention and ball simulation issues in a Distributed Environment..</i>	171

Synthetic Variable for ball position.....	172
Synthetic Variables for Collision Prevention.....	174
SUMMARY	180
CHAPTER 13: PERFORMANCE ANALYSIS.....	181
OVERVIEW.....	181
STRATEGY FOR ANALYSIS.....	183
<i>Performance with the Humanoids</i>	184
Collision Prevention Internals	184
Computer Network used for the Experiment	184
Analyzing the Results	187
<i>Performance with the 'Perf' program</i>	195
State machine of the run time system	196
What is Measured.....	198
Analyzing the Results	201
Frame Validity and Statistical Skew	216
Modified Algorithm for the Workers.....	228
Throughput in DLoVe	253
Asynchronous Frame Rate in DLoVe	257
SUMMARY OF RESULTS	259
CHAPTER 14: EVALUATION.....	260
OVERVIEW.....	260
CONCEPTUAL APPLICABILITY	261
SCALABILITY ISSUES.....	261
EXTENSIBILITY ISSUES	263
PRESERVING INTELLECTUAL INVESTMENTS.....	264
CHAPTER 15: OPEN PROBLEMS AND FURTHER STUDY	265

OVERVIEW.....	265
ENHANCING AND IMPROVING DLOVE'S PARADIGM	266
CHAPTER 16: CONCLUSION	273
APPENDIX A.....	275
APPENDIX B.....	281
APPENDIX C.....	285
APPENDIX D.....	290
APPENDIX E.....	292
BIBLIOGRAPHY	297

List of Figures

CONSTRAINT A	33
STAR GRAPH (LEFT), AND TREE GRAPH (RIGHT).....	43
LINKS AND VARIABLES	62
A DISABLED LINK.....	63
COMBINATIONS OF VARIABLES ATTACHED TO A LINK.....	64
COMBINATIONS OF LINKS ATTACHED TO VARIABLES.....	64
A VARIABLE USED FOR OUTPUT BY MULTIPLE LINKS.....	65
STATE DIAGRAM OF THE EVENT HANDLER IN THE FACTORY EXAMPLE.....	68
DATA-FLOW GRAPH.....	70
DATA-DEPENDENCY GRAPH	71
DLOVE CONSTRAINT GRAPH	71
DATA STRUCTURE THAT HOLDS THE LINKS AND VARIABLES	72
ANIMATION OF THE SOLVE ALGORITHM.....	80
A SECOND EXAMPLE OF THE SOLVE ALGORITHM	82
A THIRD EXAMPLE OF THE SOLVE ALGORITHM	83
COORDINATOR AND WORKERS ON A LAN.....	86
DLOVE'S MANAGERS	88
PARTITION INTO TWO SUB-GRAPHS	89
MULTIPLE WORKERS CONNECTED TO A COORDINATOR.....	92
THE L_COUNTER OF THE VARIABLES.....	97
PARTITIONED GRAPH TO TWO WORKERS	101

PARTITIONED GRAPH AFTER THE OPTIMIZATION ALGORITHM IS RUN ON THE WORKERS 101

MULTIPLE WORKERS CONNECTED TO MULTIPLE COORDINATORS 107

DISTINGUISHING INPUT DEVICES 109

DLOVE'S CONNECTION PROTOCOL..... 118

WORKER CONNECTS TO MASTER COORDINATOR USING DYNAMIC BINDING 120

SLAVE COORDINATOR CONNECTS TO MASTER COORDINATOR 120

SLAVES START LISTENING FOR CONNECTIONS, MASTER PARTITIONS THE GRAPH, AND THE
WORKERS ARE REQUESTED TO CONNECT TO SLAVES 122

WORKER CONNECTS TO SLAVE COORDINATOR USING DYNAMIC BINDING..... 123

SLAVE COORDINATOR PARTITIONS THE GRAPH (JUST FOR VERIFICATION)..... 123

WORKER IS READY TO PROCESS REQUESTS AND SO IT NOTIFIES THE COORDINATORS 124

REQUESTS MADE INTO MESSAGES..... 126

MULTI-MESSAGE CONTAINING MULTIPLE MINI-MESSAGES..... 128

GETE REQUESTS EMBEDDED INTO THE MULTI-MESSAGE 130

SNAPSHOT OF THE 'ARMS' PROGRAM..... 134

THE CURSOR OBJECT..... 135

THE ARM1 OBJECT 136

THE PARTS OF AN ARM1 OBJECT..... 137

THE PARTS OF THE ARM2 OBJECT 137

LINKS AND VARIABLES OF THE ARM2 OBJECT..... 138

ARMSLAVE OBJECTS WIRED TO AN ARM2 OBJECT 139

HALF ARMSLAVE OBJECTS ATTACHED TO FIRST TIP OF AN ARM2 OBJECT AND THE OTHER HALF
TO THE SECOND TIP 140

ARMSLAVE OBJECTS EVALUATED IN PARALLEL 141

TWO-USER INTERFACE IN THE 'ARMS' PROGRAM 143

ONLY ONE USER CONTROLLING AN ARM2 OBJECT..... 143

STATE TRANSITION DIAGRAM OF THE EVENT HANDLE OF AN ARM1 OBJECT AND THE CODE THAT IS EXECUTED BASED ON THE STATE TRANSITION	145
INCONSISTENCY ISSUE IN A MULTI-USER VE	147
HAND MOVEMENT FOR EYE-SELECTED OBJECT MANIPULATION.....	151
TWO HAND POSITIONS CONTROLLING THE POSITION OF AN EYE-SELECTED OBJECT.....	152
BRIGHTNESS AND EYE-SELECTION OF OBJECTS.....	153
STATE TRANSITION DIAGRAM OF THE 2ZOOM APPLICATION	154
SNAPSHOT OF THE 2ZOOM APPLICATION.....	155
A HUMANOID AND ITS PARTS	157
ACTIONS OF A HUMANOID BASED ON THE VIRTUAL BALL'S POSITION.....	158
A SNAPSHOT OF A HUMANOID THAT ITS HEAD FOLLOWS THE TRAJECTORY OF THE BALL	159
A HUMANOID AT THE POINT OF HITTING THE BALL	159
AN ORBITING OBJECT.....	161
THE SLIDER OBJECT.....	161
SNAPSHOT OF A SLIDER CONTROLLING AN ORBITING OBJECT	162
SLIDER MODIFIES SPEED OF AN ORBITING OBJECT.....	163
SLIDER WIRING IN A TWO-USER ENVIRONMENT	164
THE VIRTUAL BALL	167
A HUMANOID OBJECT.....	168
ISSUES IN PARTITIONING A HUMANOID'S GRAPH.....	169
CORRECTING THE PARTITION INEFFICIENCY	170
THREE HUMANOIDS ASSIGNED TO BE SIMULATED BY THREE DIFFERENT WORKERS	171
SYNTHETIC VARIABLES FOR THE POSITION OF THE BALL.....	173
USE OF SYNTHETIC VARIABLE FOR COLLISION DETECTION.....	175
THE DLOVE VIRTUAL PARK	177
MASTER COORDINATOR USING AN HMD AND THE POLHEMUS TO INTERACT	178
SLAVE COORDINATOR USING THE MOUSE TO INTERACT.....	179

THE SGI EXPERIMENT	185
MOUNTING PERFORMER RUN TIME LIBRARIES	186
COMPARING THE NUMBER OF EVALUATIONS OF THE NON-DISTRIBUTED AND THE DISTRIBUTED VERSION OF THE VIRTUAL PARK APPLICATION USING COMPUTATIONAL EXPENSIVE EVALUATIONS. 'SIMULATELINK' IS $O(N^3)$ WHERE N IS THE NUMBER OF HUMANIDS (32).	189
'SIMULATELINK' IS $O(N^4)$ WHERE N IS THE NUMBER OF HUMANIDS (32).	189
NUMBER OF FRAMES WHEN THE RUNNING TIME OF 'SIMULATELINK' IS $O(N^3)$, WHERE N IS THE NUMBER OF HUMANIDS.....	190
NUMBER OF FRAMES WHEN THE RUNNING TIME OF 'SIMULATELINK' IS $O(N^4)$, WHERE N IS THE NUMBER OF HUMANIDS.....	191
COMPARING NUMBER OF EVALUATIONS OF THE NON-DISTRIBUTED AND THE DISTRIBUTED VERSION OF THE VIRTUAL PARK APPLICATION USING COMPUTATIONAL INEXPENSIVE – $O(N)$ – EVALUATIONS.	193
COMPARING THE FRAME RATE OF THE NON-DISTRIBUTED AND THE DISTRIBUTED VERSION OF THE VIRTUAL PARK APPLICATION USING COMPUTATIONAL INEXPENSIVE – $O(N)$ – EVALUATIONS	194
DLOVE GRAPH OF THE 'PERF' APPLICATION	196
PARTITION OF THE DLOVE GRAPH OF THE 'PERF' APPLICATION	196
INTERNAL STATE MACHINE OF DLOVE WHEN EXECUTED IN THE DISTRIBUTED MODE	198
MEASUREMENT OF DLOVE	199
MEASURING LATENCY IN THE INTERNAL STATE MACHINE	200
THROUGHPUT OF DLOVE WITH DIFFERENT 'DROP' AND APE	201
GRAPHID(001L) DROP=30, APE=100	203
GRAPHID(002L) DROP=300, APE=100	204
GRAPHID(003L) DROP=3000, APE=100	205
GRAPHID(004L) DROP=30, APE=1000	206

GRAPHID(005L) DROP=300, APE=1000	207
GRAPHID(006L) DROP=3000, APE=1000	208
GRAPHID(007L) DROP=30, APE=10000	209
GRAPHID(008L) DROP=300, APE=10000	210
GRAPHID(009L) DROP=3000, APE=10000.....	211
GRAPHID(010L) DROP=30, APE=100000	212
GRAPHID(011L) DROP=300, APE=100000.....	213
GRAPHID(012L) DROP=3000, APE=100000.....	214
WORKERS DRIFTING IN RESPONDING BACK TO COORDINATOR.....	215
GRAPHID(001F) DROP=30, APE=100	217
GRAPHID(002F) DROP=300, APE=100	218
GRAPHID(003F) DROP=3000, APE=100	219
GRAPHID(004F) DROP=30, APE=1000	220
GRAPHID(005F) DROP=300, APE=1000	221
GRAPHID(006F) DROP=3000, APE=1000	222
GRAPHID(007F) DROP=30, APE=10000	223
GRAPHID(008F) DROP=300, APE=10000	224
GRAPHID(009F) DROP=3000, APE=10000	225
GRAPHID(010F) DROP=30, APE=100000	226
GRAPHID(011F) DROP=300, APE=100000	227
GRAPHID(012F) DROP=3000, APE=100000.....	228
GRAPHID(101L) DROP=30, APE=100	229
GRAPHID(102L) DROP=300, APE=100	230
GRAPHID(103L) DROP=3000, APE=100	231
GRAPHID(104L) DROP=30, APE=1000	232
GRAPHID(105L) DROP=300, APE=1000	233
GRAPHID(106L) DROP=3000, APE=1000	234

GRAPHID(107L) DROP=30, APE=10000	235
GRAPHID(108L) DROP=300, APE=10000	236
GRAPHID(109L) DROP=3000, APE=10000.....	237
GRAPHID(110L) DROP=30, APE=100000	238
GRAPHID(111L) DROP=300, APE=100000.....	239
GRAPHID(112L) DROP=3000, APE=100000.....	240
GRAPHID(101F) DROP=30, APE=100	242
GRAPHID(102F) DROP=300, APE=100	243
GRAPHID(103F) DROP=3000, APE=100	244
GRAPHID(104F) DROP=30, APE=1000	245
GRAPHID(105F) DROP=300, APE=1000	246
GRAPHID(106F) DROP=3000, APE=1000	247
GRAPHID(107F) DROP=30, APE=10000	248
GRAPHID(108F) DROP=300, APE=10000	249
GRAPHID(109F) DROP=3000, APE=10000	250
GRAPHID(110F) DROP=30, APE=100000	251
GRAPHID(111F) DROP=300, APE=100000	252
GRAPHID(112F) DROP=3000, APE=100000.....	253
THROUGHPUT OF DLoVE UTILIZING THE NEW ALGORITHM ON THE WORKERS	254
COMPARING THROUGHPUT BETWEEN THE ORIGINAL AND THE NEW ALGORITHM	255
COMPARING THROUGHPUT BETWEEN NON-DISTRIBUTED AND DISTRIBUTED VERSION OF 'PERF'	256
ASYNCHRONOUS RENDERING IN DLoVE	258
'ROOT' WORKER INCORPORATED IN DLoVE TO HANDLE SYNTHETIC VARIABLES IN CYCLIC CONSTRAINT GRAPHS	268
NODE CONNECTION WITH THE 'ROOT' WORKER	269

PERFORMANCE MEASUREMENT OF A 100MB SWITCHED NETWORK USING MULTIPLE
COORDINATORS..... 272

A SIMPLE VR PROGRAM CREATED USING THE DLOVE'S PARADIGM 289

DLoVe

(Distributed Links over Variables evaluation)

**A specification paradigm for
designing distributed VR applications
for single or multiple users**

Chapter 1: Introduction

Overview

For the last 20 years, desktop systems have been continuously enhanced, providing the user community with such tools as line and raster graphics, window-icon-mouse-pointer graphical user interfaces, and advanced multimedia extensions. With the help of immersive virtual environments, users now have access to 3D (3 dimensional) space.

The Responsive Workbench is an example of a 3D space environment. It is a virtual environment that allows the users to locate virtual objects and control tools on a real “workbench”. The virtual objects are projected onto the surface of the workbench. This virtual environment duplicates the actual work environment of an operating room, an architect’s office, etc, where several people can work together in this environment on a common task [Krueger 94].

The state of practice in user-interfaces today is the familiar direct manipulation, GUI (Graphical User Interface), or WIMP (Window, Icon, Menu, Pointer) style interface [Shneiderman 92]. The next generation's computer interfaces have been called non-WIMP [Green 91] and are characterized by parallel and continuous interactions. Examples are virtual reality and virtual environments [Foley 87]. Existing languages and models, event-based software, methods and tools do not satisfy the next generation requirements. For that reason we need a new model and framework for describing and implementing these interfaces from the point of view of the user and the dialogue.

Most of today's examples of non-WIMP interfaces have been designed and implemented with event-based models that are more suited to previous interface styles. Because there is no software that can describe continuous, parallel interaction explicitly (which is needed for virtual reality) and the old models (event-driven) fail to capture continuous, parallel interaction explicitly, new interfaces have required considerable low-level programming. While some of these interfaces are very inventive, they have made such systems difficult to develop, reuse, and maintain.

DLoVe (Distributed Links over Variables evaluation) is my new model for next generation dialogues. This model expresses non-WIMP formal structure in the same way that existing technology expresses command-based, textual and event-based dialogues. My model combines a data-flow or constraint-like component for the continuous relationships with an event-based component for discrete interactions. The modules describing the continuous relationships between objects form a constraint graph, which can be evaluated by DLoVe's constraint engine. The constraint engine ensures that all relationships between the related objects are

satisfied. Individual continuous relationships can be enabled or disabled on the fly. Programs designed in this framework, not only capture continuous and parallel interaction explicitly, but also allow programs to run in parallel when there is a need for faster computation. In addition, this framework allows programs to run in a multi-user environment where users can participate in a collaborative environment.

In DLoVe, one can write programs designed for a single machine but can execute them in a distributed environment without any code modification. The majority of research done in parallel systems has concentrated either upon computational models, parallel algorithms, or machine architecture. By contrast, little attention has been given to software development environments or program construction techniques required in order to translate algorithms into operational programs [Sunderam 90].

For implementing this framework, DLoVe assigns roles to different machines. For example, one machine, the Coordinator, is responsible for rendering the graphics on the screen and reading all input devices, and all other machines, the Workers, are responsible for keeping the constraints between Variables up-to-date and serving requests to the Coordinator. As a result, the Coordinator has more time to spend refreshing the screen and providing an immersive environment to the user, because the responsibility of keeping all the Variables up-to-date is taken away from it and given to other machines that are dedicated to this task.

All machines/processes involved in running a program in parallel have an identical copy of the constraint-graph so that every machine/process works on the same constraint graph. However, each machine/process is only responsible for evaluating part of the constraint graph. This way, the constraint graph seems to be distributed

among many machines/processes. Because every machine is responsible for only a sub-graph of the constraint graph, the Coordinator can request services in parallel from the machines that are dedicated to parts of the constraint graph. In other words, by distributing the constraint graph and then partitioning it, so that, different machines are responsible for different parts of the constraint graph, the queries get partitioned. By having the queries partitioned, the Coordinator can request services in parallel where some queries always go to machine 'A' others to machine 'B' and so on.

Ease of Use

Ease of use is determined by how fast and effortlessly the application can be learned and used by users as well as by counting the frequency of errors when interacting with the application. Often, applications that are robust, functional, and exhibit good performance, are abandoned by the user community due to difficult or foreign user interfaces. The success of an application is largely determined by how easily the application can be learned and used [Larson 92].

When using the programming paradigm of DLoVe, the programmer uses a production programming language with which he/she is familiar. One can use, for example, C++ to write programs as he/she formerly wrote for a single machine and a single user. However, DLoVe gives the programmer the flexibility to run the same program in parallel if he/she wishes, without any additional programming effort, making DLoVe a desirable model for describing Virtual Reality Environments and behavior. When such a program is transformed to a multi-user program the

programmer must manually make large, but trivial, revisions to the program. However, its base form will remain same.

To transform a single-user program that runs sequentially to a program that runs in parallel, no modifications are required. The user can just recompile using a different set of libraries. To transform a single-user program to a multi-user program, some trivial modifications must be made. In a multi-user environment we need to designate and differentiate between the input and output devices so we can give different roles to different users. Also, the users must be able to see each other in the virtual environment, whereas in the single-user environment there is only one user and one view of the virtual world.

By using DLoVe, programmers can write networked and distributed virtual reality programs without any knowledge of networked, distributed, or parallel programming, making Virtual Reality programming quick, simple and fun.

Outline of the Dissertation

Chapters 2 through 5 describe background and related work. Chapter 2 describes Virtual Reality, Virtual Environments, and the basic problem of real time rendering that DLoVe solves. Chapter 3 describes constraint graphs and constraint engines that define DLoVe's framework and programming paradigm. Chapter 4 describes techniques used in programming Parallel and Distributed systems, which motivate DLoVe's approach to parallelism. Chapter 5 describes multi-user interfaces, and

builds a framework for understanding virtual environments and multi-user programming.

Chapters 6 through 9 describe the DLoVe programming. Chapter 6 describes Links and Variables of the DLoVe underlying framework, and its incremental constraint solver that keeps the constraint graph up-to-date. Chapter 7 describes how a program written in DLoVe that runs on a single machine can be transformed to run in parallel on multiple machines, and how this is accomplished. Chapter 8 describes Parallel computation in DLoVe, where one can describe a program that runs on a single machine for a single user, which allows programs to run on multiple machines for multiple users. Chapter 9 describes all methods that the DLoVe protocol supports, how all machines connect to each other, how the Partition algorithm works and the good and bad points of each of the methods used to run a parallel program in DLoVe.

Chapters 10 through 12 demonstrate the use of DLoVe's paradigm in creating Virtual Reality applications. Chapter 10 describes in detail the creation of a simple VR program (arms). Then it describes how such a program can be executed in a distributed environment. Finally, it describes the transformation of the arms program to be executed in a multi-user environment. Chapter 11 describes an application that uses multiple input devices such as Polhemus and an eye tracker device to implement the 2zoom application where a user can select and manipulate remote objects. Chapter 12 describes a large-scale application that involves interaction between multiple users and multiple computer-simulated entities, call Humanoids. The virtual environment is a Virtual Park where the Humanoids play with a virtual ball. The simulation requires execution of computationally expensive

algorithms for the animation of the Humanoids and the interaction between them as well as between the users.

In chapters 13 I present the results of the performance measurements. Traditional measurements failed to describe accurately the performance of DLoVe. New algorithm and ideas are presented in this chapter for measuring performance in DLoVe that describe it accurately.

Chapter 14 evaluates DLoVe and describes its applicability and limitation. Chapter 15 presents the open problems and future work that needs to be conducted to both improve DLoVe's paradigm and also its performance. Chapter 16 is a summary of this dissertation.

Chapter 2: Related Work - Virtual Reality

Introduction

DLoVe is, primarily, a mechanism for creating VR systems. Virtual Reality (VR) eliminates the need to work with a flat 2D image, the keyboard, mouse and monitor. All of these interface methods have become very familiar to the users, but they still remain unnatural and limiting. VR, on the other hand, allows the user to interact in a virtual world naturally, as he/she would interact in the physical world. Real time 3D interaction allows the user to touch, feel, and lift objects as he/she would in the physical world. VR visually isolates the user from the physical world and substitutes an imaginary 3D. This imaginary world is constructed using computer-generated images displayed to the user through a head-mounted display (HMD) (or some other immersive display devices) and a spatial interaction device such as DataGlove or PowerGlove (Satava, 1993, pp. 203-05).

Virtual Reality is a breakthrough technology that alters the way users interact with computers and it consists of three major elements: interaction, 3-D graphics, and immersion [Prat 95]. Making a virtual environment realistic is not enough; that environment must also allow the user to interact with it in real time in order for it to be called a Virtual Reality environment [Stephen 94] [Barfield 95]. The goal of a VR developer is to let the user focus on the virtual model or environment, to disregard everything else [Hodges 95]. Clicking and dragging might be interactive, but it is not VR because it is not immersive. VR lets the user manipulate and navigate through the virtual model in real time. VR is available in amusement arcades, films of the future, and is widely used for research purposes, and lately for industrial settings.

Why use Virtual Reality

The great benefit of using Virtual Reality is the ability to work in a virtual environment without the attendant danger, impracticality, or significantly greater expense that would be encountered in the same environment if it were physical. That means saving money and time, and enhancing creativity, in product prototyping, hazardous task training, molecular modeling, medical education, entertainment content creation, and a range of other mission-critical tasks. VR adds value to virtually any application where it is vital to experience spatial relationships, and analyze, design, engineer and understand such relationships.

Today, VR is used in many different areas in research that include manufacturing, architecture, medicine and healthcare, entertainment, scientific data visualization,

the military and many more. For example, a surgeon can perform surgery from a remote site, or the surgeon may perform surgery on a virtual patient first before performing it on the actual patient, for practice or training purposes [Bowman 95].

Researchers at Georgia Institute of Technology's Graphics, Visualization, and Usability Center are using VR for therapy of patients with psychological disorders such as acrophobia. The patients were moved up on high elevators in a Virtual Environment, or given the illusion of looking through a third floor window, but they were physically in a safe environment, a lab [Hodges 95]. This way, patients could overcome the fear of heights and adjust more easily to real world situations.

Using VR designers are able to design large structures, work in them and make modifications and strategic changes to these structures. For example, in the Virtual Factory Project, researchers in Iowa State University designed a factory where robots replace finished parts and move along virtual tracks. This helps manufacturers investigate new technology and architects re-design structures before they are built, because modifying a structure after it is built is expensive and time consuming. [Kelsick 98]

Limitations in VR

Traditionally, computers use input devices ranging from switches and punched cards to keyboards and mice. However, such devices are insufficient for specifying actions in a virtual world. For example, how would you express the simple act of drinking from a cup of water by using a mouse? Typing a keyboard command may

come to mind, but this quickly becomes cumbersome, as you will have to specify which cup to drink from, and how much to drink. You will also have to learn the correct syntax to convey the information to the computer. This is definitely not a simple task [Vince 95][Burdea 94]. That is why new input devices needed to be created, such as 6D mice, joysticks, wands, force balls, DataGloves, and PowerGloves that use Polhemus [Aukstakalnis 92] to describe their location and orientation in space. A Dexterous Hand Master (DHM) is an exoskeleton that is attached to the fingers using velcro straps. Attached to each finger joint is a device called a Hall effect sensor whose purpose is to measure the finger-joint angle. Tod Machover used an Exos DHM at the MIT Media Lab to control acoustic parameters in live musical performances [Sturman 92].

The ability to feel objects in virtual environments can markedly enhance the effectiveness of many applications, particularly for training, scientific visualization, and telepresence [Langreth 95]. Haptic devices for presenting tactile and force sensations, are being developed in several laboratories, but are not yet widely used elsewhere. For example, Makato Sato of the Tokyo Institute of Technology's Precision and Intelligent Laboratory has developed a force-reflecting system called Spidar (Space Interface Device for Artificial Reality). With this system, the user inserts his or her thumb tips and index finger into a pair of rings, each of which have four strings attached to rotary encoders. String movements are restricted with breaks, providing touch sensations [Bowman 95].

The visual sense is used more than any other to process information. Even a quick glance at an intricate picture will be enough to process most of the details of the scene. Visual input is a necessary requirement for an engaging virtual environment.

Stereoscopic vision occurs when two separate images are generated and viewed. The left eye views the left image and the right eye views the right image. The two images differ slightly due to visual parallax, caused by the distance between the eyes [Vince 95] [Burdea 94].

To help users immerse in a virtual environment headmounted (HMD) displays are being used where two monitors or LCD displays are always in front of the eyes. Other output device include Projected HMD and Mini HMD, Heads-up displays which can then overlay the “real” world with additional information, thus augmenting the “real” world. Virtual reality can be used to enhance actual reality [Adam 93] [Doyle 95].

Different input and output devices are suitable for different tasks, but one requirement for any device to work well in VR is that the lag between the action and what appears on the screen cannot be too great. For example, if you move your DataGlove and the direction is changed only 5 seconds later, you can be easily confused and the device becomes hard to use.

VR systems have one important limitation, the inability to move around the virtual environment in a natural way. An observer is either constrained by the physical boundaries, as with the CAVE system [Bowman 95], or by the range of a head tracking system.

The CAVE system (Cave Automatic Virtual Environment) developed at the Electronic Visualization Laboratory of the University of Illinois, Chicago, uses stereoscopic video projectors to display images on three surrounding walls and on the floor. The participants wear glasses with LCD shutters to view the 3D images [Bowman 95].

Physiological dangers of VR

Problems arise because of an important phenomenon of latency between head movements and the image that the computer sends to the HMD. In this case, the human brain receives conflicting information from the involved senses. Since the represented image does not truly reflect the movement perceived by the semicircular canals, the brain cannot decide which is the right information.

Another cause of conflict is the difference between the convergence of the eyes and the convergence created by the pair of stereoscopic images. In everyday life, we are constantly forced to focus on objects at distances from 28mm away to infinity. An HMD helmet has two liquid crystal screens that are essentially two-dimensional devices. The two images are merged to create a stereoscopic illusion. In spite of everything, there is a difference between the stereoscopic image created by the HMD helmet, and that perceived in reality. In the case of an HMD helmet, the two images are presented with the aid of a combination of optical lenses. This creates a virtual image at a fixed distance, which is located from several meters ahead to infinity, according to the model of HMD helmet used. As a result, the user always focuses at a fixed distance, which is something that is not happening in real life.

There is the notion of conflict due to contradictory information. Each time there is a conflict, there will be an effect perceived by the human body. These observations must not be taken lightly. Certain types of trauma are very pronounced, although they have a short duration. For example, latency caused by an HMD helmet, a

glove, or a slow computer can cause some to experience heart rate increase, seasickness, and vomiting. Others are unable to focus adequately once they return to the real world, and after a while can regain some hours, according to the length of the immersion [Reason 78].

Chapter 3: Related Work - Constraints and Constraints in UIMSSs

Introduction

Since the earliest interactive graphical editors, systems have attempted to provide sophisticated editing features. Sketchpad [Sutherland 63] introduced not only direct graphical interaction, but also constraints and snapping¹. Constraints are useful in programming languages, user interface toolkits, databases, database queries, simulation packages, and other systems, because they allow programmers or users to state declaratively a relation that is to be maintained, rather than requiring them to write procedures to maintain the relation themselves.

¹ Snapping is a method that allows users to select, align, etc. objects on a grid or a virtual gravitational force. For example, when a user wants to select a line in a drawing application, the user can click near the line and the application selects the line without the need of the programmer to click right on the line, enhancing his/her selection skills. Snapping is implemented in numerous applications. Another application is to help users align objects on a grid. The users draw and drag objects, and the application always snaps the objects on a virtual grid. [Bier 86]

Maintaining that relation is left up to the underlying system, instead of being the responsibility of the programmer. A constraint typically contains both a declarative description of the relation and a set of procedures for making that relation hold, which are used by the underlying constraint satisfier/engine. The emphasis on constraints helps define a new style of programming, one in which the focus is on computing data values instead of writing methods [Myers 92b].

Any large, complex application contains hundreds, even thousands of inter-dependent relationships. For example, a graphical application must deal with the relationships arising from moving objects on the screen, displaying feedback when the user moves/deletes objects, keeping text labels on buttons centered, displaying an alarm on the screen when the reactor gets overheated, and in general keeping the view consistent with the data they represent. Constraints provide a convenient way to specify relationships and have them automatically maintained at run time by the constraint solver/evaluator. In contrast, in a traditional programming language constraint handling is not even specified.

For example, suppose that a slider indicates the speed of a rotating object. When the slider is at the top, the rotating object rotates at maximum speed, and when the slider is at the bottom, the rotating object rotates at minimum speed. In constraint programming, wherever the application/user changes the slider's position, the constraint solver automatically changes the speed of the rotating object. In contrast, a conventional language forces the designer to write code to figure out where the slider is, and change the speed of the rotating object. This might not seem such a difficult task, but when an application contains thousands of such relationships, the bookkeeping needed to maintain them increases so rapidly that

adding new functionality to the application becomes difficult, and the time required to debug the changes also increases substantially.

Evolution of Constraints

In addition to simplifying the creation of applications and increasing their robustness, constraints lend themselves to incremental re-computation/re-evaluation. When a user changes one or more parameters in an application, or adds or deletes a number of constraints, most of the existing constraints remain satisfied and only a small number must be re-evaluated. An incremental constrained-solving algorithm, such as Eval/vite [Hudson 91] and the algorithms presented in [Zanden 94], can automatically identify which constraints must be reevaluated and limit its solving to these constraints. Such an algorithm can be used with any application written for that constraint system. In contrast, a conventional language requires the designer to create a new incremental algorithm for each application. Of course, a conventional language also lets the designer take advantage of any special characteristics of the application, so that he or she can write a custom algorithm that might be faster than a general-purpose constraint solver.

One-way and Multi-way Constraints

Constraints can be either one-way or multi-way. One-way constraints differentiate variables as being inputs or outputs. When there is a change in the input variables,

the change propagates to the output variables so that the relationships are all satisfied. If there is a change in the output variables however, the change does not propagate backwards to the input variables. In contrast, in multi-way constraints if there is a change in either input or output variables, the change propagates to the variables on the other side of the graph. All variables are treated as both inputs and outputs.

For example, using the example above with the slider and the rotating object, if the slider moves the rotating object also changes speed with the respect to the slider's value. However, if the speed of the rotating object changes via another input and not the slider, the slider does not move, in a one-way constraint solver, to correspond to the newly changed speed of the rotating object. However, in a multi-way constraint solver, the slider would have changed to reflect the corresponding speed of the rotating object.

Multi-way constraints are obviously more powerful than one-way constraints, but this increased expressiveness comes at a price. One-way constraint solving algorithms only have to evaluate constraints in one direction, making them simpler to implement and more efficient than multi-way constraint solvers that must also choose which variable in a constraint should be modified.

Multi-way constraints can also introduce ambiguity at the design level. For example, suppose we have the constraint $A - B - C = 0$. If "A" changes, the constraint solver must choose whether to change "B", "C", or both. One approach to eliminate this ambiguity is to divide constraints into hierarchies, or assign priority levels, or strengths to constraints [Sannella 94] [Freeman-Benson 90] [Lopez 94a] [Lopez 94b]. The constraint solver then tries to satisfy as many constraints as

possible, solving the highest priority constraints first, and then the next highest, and so on. One issue that constraint hierarchies do not address is how to specify a constraint when multiple values should change. For example, what happens when “A” changes, should both “B” and “C” change, or should just “B” change, or should just “C” change? Given a constraint hierarchy, where constraints have associated strengths, a constraint solver may leave weaker constraints unsatisfied in order to satisfy stronger constraints.

In addition to constraints of varying strength or priorities, Kaleidoscope [Lopez 94a] [Lopez 94b] has constraints of varying duration. Constraint duration specifies the period of validity for constraints. The most flexible model would allow constraints to be asserted and retracted at arbitrary points in time. However, this would lead to difficulties in predicting behavior, since any piece of code could have a side effect on which constraints are active. In Kaleidoscope the default constraint duration is ‘always’, which causes a constraint to remain active forever. A ‘once’ duration instructs the system to assert the constraint causing it to be enforced at that moment (and thus potentially affecting values), and then immediately retracts it. Finally, the ‘assert/during’ construct specifies that a constraint should remain in force during the execution of a block or loop.

Lazy and Eager Constraint Solving

Constraint solving can be in either a lazy or eager style [Zanden 94]. Lazy evaluation (used by Garnet [Myers 90a] [Myers 90b], Eval/vite [Hudson 91] and other systems, evaluates a constraint only if it affects a result that the user

requests. Eager evaluation evaluates a constraint immediately when values change. Thus, a lazy-evaluation system can obtain variables whose values are out of date. Lazy evaluation avoids unnecessary work if relatively few values are needed to compute the result the user requests. However, lazy evaluation also introduces extra bookkeeping, since the constraint solver must keep track of out-of-date variables. In addition, lazy evaluation can result in potential delays when the values of out-of-date variables are demanded. Lazy evaluation is most effective in applications where the user wants to view only a limited portion of the application's data and where changes are occurring to all parts of the application's data. Otherwise, eager evaluation is preferable, since it is conceptually cleaner than lazy evaluation because all variables are almost always up-to-date.

Constraint Systems

People have developed constraint solvers that support both lazy and/or eager evaluations, incremental solving and one-way and multi-way constraint solvers. For example, the DeltaBlue [Freeman-Benson 90] algorithm (an incremental version of Blue) and SkyBlue [Sannella 94] (a more general successor of the DeltaBlue) maintain an evolving solution to the constraint hierarchy as constraints are added and removed. DeltaBlue minimizes the cost of finding a new solution after each change by exploiting its knowledge of the last solution.

Both DeltaBlue and SkyBlue exploit their knowledge by associating sufficient information with each variable to allow the algorithm to predict the effect of adding a

given constraint by examining only the immediate operands of that constraint. This information is called the ‘walkabout strength’ of the variable, defined as follows:

“Variable v is determined by methods m of constraint c . v ’s walkabout strength is the minimum of c ’s strength and the walkabout strengths of m ’s input.” [Freeman-Benson 90]

The walkabout strength is the weakest constraint that can be revoked; thus weaker walkabout strengths propagate through stronger constraints.

There are constraint solvers that support cycles in the constraint graph, such as SkyBlue [Sannella 94], and others that do not, such as DeltaBlue [Freeman-Benson 90]. To satisfy cycles, SkyBlue calls external cycle solvers based on Gaussian elimination to satisfy the constraints around the cycle, and uses local propagation to satisfy the rest of the constraints. If the available cycle solvers cannot satisfy the constraints in a cycle, the variables are marked invalid, so the programmer can tell when the cycle constraints are not satisfied.

Both DeltaBlue and SkyBlue support required and preferential constraints. The required constraints must hold. The system should try to satisfy the preferential constraints if possible, but no error condition arises if it cannot. SkyBlue and DeltaBlue allow an arbitrary number of levels of preference, each successive level being weaker than the previous one. The set of all constraints, both required and preferred, labeled with their respective strengths, is called the constraint hierarchy. [Freeman-Benson 90][Sannella 94]

Many systems have been designed based on these powerful and flexible constraint solvers. For example, the Multi-Garnet package [Sannella 92] uses SkyBlue to add

support for multi-way constraints and constraint hierarchies to Garnet [Myers 90a]. Garnet is a user interface toolkit built on Common Lisp and X windows with emphasis on handling objects' run-time behavior and on handling all visual aspects of a program's user interface, including its graphics and the contents of all application-specific windows [Myers 90b]. Multi-Garnet constraints support many of the useful features of Garnet's one-way constraints, including indirect references to constrained object slots through a series of other slots and inheritance of constraints in Garnet's prototype-based system.

TBAG [Elliott 94] is a toolkit for creating interactive 3D graphics that uses constraints to maintain relationships between time-varying properties of graphic objects such as their positions and the derivatives of their positions. These relations may be given different strengths since TBAG uses SkyBlue to maintain the constraints.

The VB2 [Gobbetti 93] Virtual Reality system also uses SkyBlue to maintain connections between 3D input devices and objects in the virtual world, and to attach virtual tools to objects that the user is editing. These constraints can be added or removed as the user manipulates different virtual objects in the virtual world.

The Kaleidoscope language [Lopez 94a] [Lopez 94b] integrates constraints and imperative, object-oriented programming. The current implementation of this language (Kaleidoscope'93) uses SkyBlue to maintain primitive constraints.

Alan Borning [Borning 86] describes a bi-directional system where the constraints can be Static or Temporal, and can be designated as Reference only, or Anchor.

A Static constraint describes a relation that must hold at all times. Static constraints are similar to required constraints in SkyBlue and DeltaBlue. Each such constraint is represented as a predicate, which may be used to test if the constraint is satisfied, and a set of methods that can be invoked to satisfy the constraint. If any of the methods are invoked, the constraint will be satisfied. Optionally, a constraint may also have an error method that returns a real number indicating how nearly the constraint is satisfied.

Temporal constraints have a somewhat different representation. Two sorts of temporal constraints are provided: those that describe continuous evolution of objects - time function constraints and time differential constraints - and those for discrete evolution - trigger constraints. Time, as in TBAG [Elliott 94], is a distinguished variable in several respects and may not be effectively treated as an ordinary variable in a standard constraint relation.

Each kind of constraint is satisfied differently. Temporal constraints must be satisfied globally, in contrast to the normal local satisfaction of static constraints. In addition, normal constraint relations have the property of satisfying themselves in multiple directions, depending on computational circumstances.

Reference-only constraints include additional instructions on how they may be satisfied. Whether or not the constraint is satisfied is partially determined by the value(s) of the attached input variable(s), but the variable(s) may not be altered to satisfy the constraint.

Anchor constraints may be used to specify that an object's value is fixed. The reference-only and anchor designations are related but not the same. If a variable is designated as reference-only by a particular constraint, that constraint may not alter the variable, but some other constraint can. If a variable is anchored, no constraint may alter it. [Borning 86]

Kaleidoscope'93 [Lopez 94a] [Lopez 94b] combines constraint and object-oriented programming while preserving a familiar object model from imperative programming. Objects have states and methods, as in most object-oriented languages. Constraints may be placed between objects and object slots, and once a constraint is established, the system attempts to enforce the constraint by filling slots with values. As objects change by assignment, these long-lived constraints re-execute and find new values for their slots. Similar to methods, these constructors are able to reference variables indirectly through many levels of pointers. Indirect references were the key extension to constraints, which allowed Garnet to be the first comprehensive user interface toolkit to be built on top of a constraint system [Myers 92a] [Myers 90a]. The success of indirect constraints in Garnet has inspired their use in many other systems including MultiGarnet [Sannella 92], Rendezvous [Hill 93], Eval/vite [Hudson 91], and others.

Kaleidoscope'93 is similar to many other object-oriented languages. It has classes, objects with mutable state, methods, destructive assignments, and so forth. The key difference between constraint imperative programming and imperative programming is the ability to relate variables (such as slots/instance variables, locals, globals, etc.) by constraints. When a variable has one or more constraints on it, the constraint solver is allowed to alter the binding of the variable, or the state of the object bound to the variable, to satisfy the constraints.

Other Constraint Imperative Languages (CIPs) do not allow constraints between arbitrary objects, and restrict constraints to instance variables. For example, Siri, another CIP language that is probably the closest relative to Kaleidoscope'93, only re-satisfies constraints between instance variables within the representation of a single object [Horn 92a] [Horn 92b]. Most of these systems support a mechanism for deleting constraints at run time. This is necessary when the constraints are no longer needed. However, since most constraints will be needed in the future, deleting them and then re-creating them and updating the constraint graph may be expensive.

This is why in DLoVe constraints are never deleted. Instead, when a constraint is not needed any more, the constraint is marked as disabled. The effect of this is the same as deleting or removing the constraint but instead of recreating the constraint and attaching it to its dependencies, DLoVe only marks the constraint as enabled. This implements instantaneous removal and insertion of constraints.

Algorithms in Constraint Solving

Constraint solving algorithms are typically either data driven or demand driven. Data-driven algorithms start at the point(s) of change and propagate that change outward. Demand-driven algorithms start at the point(s) where the data are ultimately used, and work toward the point of change. Each of these approaches has advantages. Eval/vite [Hudson 91], DLoVe, and the algorithms in [Zanden 94]

use a combination of both to capture the good features of both, data and demand driven algorithms.

Data-Driven

Data-Driven algorithms start with the direct change set of a transaction, that is, the set of variables that have been modified by the operations of a transaction, and proceed by evaluating constraints that are known to need re-evaluation. The fact that a constraint needs to be re-evaluated is recorded by its membership in the “work” set. A constraint is placed in the “work” set if and only if one of its input variables has changed value. Such an algorithm may be implemented as incremental with little effort so that it does not evaluate all constraints after a change, only those whose input variables change value either directly by the user, or indirectly by the solver. Constraints whose variables are unrelated to the change are not evaluated. While a data-driven algorithm can be made efficient in some cases, it will not directly support lazy computations [Hudson 91]. It will always evaluate any constraints whose input variables change value even if the new value of that variable will never actually be used. If not all variables are needed at any given time, it may be possible to obtain significant savings by avoiding the computation of constraints whose variables are not used.

Demand-Driven

Demand-driven algorithms start with the set of variables whose values are actually needed at any given time and recursively evaluate the constraints needed to compute those values as a part of a depth-first traversal. This results in a topological ordering of evaluations. The problem with such algorithms is that they may end up evaluating constraints whose variables are not needed at the moment resulting in wasted computations.

The nullification/re-evaluation scheme

The nullification/re-evaluation scheme describes the methodology that a one-way constraint system needs to take in order to keep relationships satisfied. When the value of a variable changes, either by direct modification or by installation of a new formula/constraint in the variable, all variables that directly or indirectly depend on this changed variable are marked out-of-date (nullification phase). When the value of a variable is requested, the constraint that computes its value starts demanding the values of other variables upon which it depends (reevaluation phase). If these variables are out-of-date, they will recursively demand the values of the variables they depend on, until variables are reached whose values are up-to-date, at which point constraints can compute their values and return [Hudson 91] [Zanden 94].

Description of Algorithms for Constraint Solving

The three algorithms are presented in [Zanden 94] and Eval/vite [Hudson 91] are the algorithms that mostly inspired and challenged me to develop the DLoVe constraint solver. The first algorithm in [Zanden 94] supports lazy evaluation, the second supports eager evaluation (with no cycles) and the third supports eager evaluation with cycles.

The lazy evaluation algorithm is constructed based on the nullification/reevaluation strategy presented in [Hudson 91]. Nullification/re-evaluation algorithms were originally constructed with the assumption that the edges in the graph remain unmodified while the constraint solver is evaluating the graph. Since the pointer variables may change, indirect reference constraints can cause the graph to change dynamically while the constraints are being evaluated, causing constraints to access information from a different set of input variables. To handle this situation, this algorithm extends to the point that dependencies can be dynamically deleted as the constraints are being invalidated. This is implemented by using timestamps on both the nodes and the edges of the graph. Node timestamps represent the number of times the node has been evaluated; each time the node is evaluated, the timestamp is incremented by one. The timestamp on an edge is the value of the timestamp on the node that the edge points to at the time the dependency was either created or last updated. A dependency's timestamp is updated whenever a node requests the value of the node that the dependency originated from. A variable's out-of-date flag is set to false before a formula is evaluated to ensure that cycles terminate after one loop. If a variable is requested a second time, it will return its old value instead of trying to evaluate itself again, thus terminating the cycle.

The eager-evaluation algorithm, which does not support cycles, uses a variation of an eager evaluator developed by Hoover [Hoover 87]. This algorithm assigns position numbers to the nodes in the graph. The position numbers indicate the node's relative position in topological order, which is always less than the position numbers of any of its successors. When a node changes value, all of its immediate successors are added to a priority queue based on their position numbers. The Hoover [Hoover 87] algorithm assumes that dependency graphs cannot change once constraint evaluation begins, so the reordering scheme and the evaluator can be invoked in sequence. However, indirect reference and conditional constraints may cause the edges of the graph to change during constraint evaluation. Thus the numbers assigned to the nodes may become incorrect and force an equation to be evaluated prematurely. To overcome this difficulty, this algorithm dynamically updates the position numbers each time the graph changes, and evaluates nodes according to this revised topological order.

The eager algorithm that does support cycles is a modified version of the previous algorithm. This is done by collapsing cycles into a single node, with each of the equations in the cycle having the same position number. Variables that are not in a cycle are evaluated, as they were when cycles were not allowed. Variables in a cycle are evaluated using the nullification/reevaluation scheme.

Comparison of Constraints

While pointer variables are commonly incorporated in programming languages, they have been incorporated only recently in their full generality in constraint systems. ThingLab [Borning 81] provides a limited form of indirect reference constraints. Programmers can construct path names that allow a constraint to traverse a structure hierarchy to find an object. When one of the components in the structure hierarchy changes, the new object is automatically referenced by the constraint. However, the constraint-solving algorithm does not support arbitrary references to objects through pointer variables.

Coral also supported a restricted version of indirect reference constraints [Szekely 88]. Coral allowed designers to declare the slots of an object that could be used as pointer variables for indirect reference constraints. Designers could then define constraints that accessed objects indirectly via these variables. However, the Coral pointer variables are not completely integrated into the constraint system. Programmers have to know whether a slot of an object is a pointer variable or not, and to set it, they have to use the constraint's appropriate procedure. In addition, the values of pointer variables cannot themselves be defined using a constraint, and so they restrict applicability of the indirect reference constraints.

Eval/vite [Hudson 91] supports a model of indirect reference constraints that is somewhat more restrictive than the one presented in [Zanden 94]. Eval/vite allows constraints to be defined in a limited subset of C++ and then translated into C++ code for incremental update. Iteration is not yet supported, and constraints cannot have a variable number of inputs, which precludes writing constraints over dynamic

sets of objects. The restriction that constraints can only have a fixed number of input variables does lead to a more efficient implementation, because it is never necessary to dynamically add or remove edges from the graph. Since each variable has a fixed number of input edges, it is possible simply to adjust edges instead. For example, if a pointer variable causes a constraint to reference `'rect.right'` rather than `'circle.top'`, the incoming edge can be adjusted so that it originates from `'rect.right'` rather than `'circle.top'`.

Rendezvous [Hill 93] supports indirect constraints for both the source and targets of a constraint, permits both variable numbers of sources and targets, and allows constraints to consist of arbitrary Lisp expressions. Rendezvous uses eager evaluation, but it differs in two respects from the algorithms in [Zanden 94]. First, it does not use position numbers but instead uses depth-first search to visit and topologically order all the variables that are affected. Then it evaluates all the variables that are affected. This approach may evaluate more than the minimum possible number of variables but it does not have the overhead of computing position numbers. Second, Rendezvous will not try to evaluate a constraint until it is sure that all of the constraint's predecessors have been computed. Rendezvous knows the constraint's predecessors in advance because of programmer-written source specifications. If there is any doubt as to whether a constraint should be evaluated, the constraint becomes suspended. In contrast however, Garnet [Myers 90a] does not determine a constraint's inputs until the constraint is evaluated. Thus there is no way to predict in advance if a constraint's position number is still valid, and the constraint is evaluated on the assumption that the position number is valid. The absence of an input expression has the disadvantage of causing Garnet to start evaluating some variables prematurely. However, by detecting sources dynamically, Garnet can determine the exact set of sources used by a constraint on

each invocation. In contrast, Rendezvous identifies the sources statically, which requires that all potential sources be listed. On any given constraint invocation, the set of sources actually used may be a subset of the sources listed. For example, if a constraint has a conditional of the form:

$$d \rightarrow \text{val} = \text{if } (\text{this} \rightarrow \text{val} > 0) \text{ then } (\text{b} \rightarrow \text{val} + 10) \text{ else } (\text{c} \rightarrow \text{val} + 10)$$

Constraint A

then the potential set of sources is the set $(\text{this} \rightarrow \text{val}, \text{b} \rightarrow \text{val}, \text{c} \rightarrow \text{val})$, but the actual set of sources is the set $\{\text{this} \rightarrow \text{val}, \text{b} \rightarrow \text{val}\}$ if $\text{this} \rightarrow \text{val}$ is greater than 0, and $\{\text{this} \rightarrow \text{val}, \text{c} \rightarrow \text{val}\}$ if $\text{this} \rightarrow \text{val}$ is less than or equal to 0. For example, a change to $\text{c} \rightarrow \text{val}$ will always cause the above constraint to be reevaluated in Garnet. Eval/vite [Hudson 91] handles correctly such constraints, though. DLoVe, however, will always evaluate the constraint A if $\text{c} \rightarrow \text{val}$ changes.

Kaleidoscope [Lopez 94a], which also uses pointer variables, supports a different type of abstraction rather than procedural abstraction. Procedures (called constraint constructors) consist of a set of constraint statements and produce as output a set of constraints instantiated with the parameters passed to the procedure. The constraints may contain indirect references, such as $\text{rect} \rightarrow \text{left} = \text{object_over} \rightarrow \text{left}$. Kaleidoscope has a well-defined notion of time, as TBAG [Elliott 94] does, and at each user-directed advance of time, object_over may be rebound (internally rebinding is treated as the retraction of one constraint). Kaleidoscope can satisfy constraints using an appropriate algorithm that handles direct references, such as DeltaBlue [Freeman-Benson 90]. The pointer model presented in [Zanden 94] differs from the Kaleidoscope model in that pointer

variables are directly handled by the constraint satisfaction algorithm rather than by asserting and retracting constraints.

Continuous and Discrete Time

TBAG is a paradigm and toolkit for rapid prototyping of interactive animated 3D graphics programs, designed to support a continuous time model, like Bramble [Gleicher 93] and DLoVe. Bramble is a toolkit for constructing graphical editing application whose constraint manages non-linear constraints and maps interactive controls and constraints to object parameters. Both TBAG and Bramble provide an almost invisible syntactic interface to the continuous time model.

A constrainable represents a conceptually continuous flow of values, out of which the application (or the system) can retrieve a value corresponding to a specific time using type-parameterized functions (templates in C++), which can be interpreted as an infinite family of function declarations. As noted above, interaction that is conceptually continuous is encoded directly into constrainables, and thus the application does not need to deal with tracking events from conceptually continuous devices.

Examples of conceptually continuous interaction include, drinking from a cup in a virtual world, throwing a ball in a virtual park, and driving a car. There are however, other interactions that are fundamentally discrete (event-based). Examples include button presses and menu choices.

TBAG applications generally deal with such discrete input events by retracting some existing assertions and asserting new constraints. Bramble uses a similar mechanism [Gleicher 93]. In DLoVe I send event tokens to event handlers to enable/disable Links.

For instance, when the user's virtual hand intersects with a virtual object, the user can press the left mouse button to send an event to indicate that the virtual object should be attached to his/her virtual hand's position in space. Thus, when the user moves his/her hand, the object moves along with his/her hand. Releasing the mouse button sends another token that indicates that this hand - object relation should be terminated. Both TBAG [Elliott 94] and DLoVe were designed to provide a fundamentally continuous, rather than discrete, treatment of naturally continuous phenomena such as time and motion.

Chapter 4: Related Work - Parallel and Distributed Systems

Introduction

The need for more and more computing speed in rendering and simulating Virtual Environments has caused many people to consider use of parallel or distributed computing. In *parallel computing*, several machines or processors are devoted to solving one problem in the shortest possible time. In *distributed computing*, system resources from a network of general-purpose computers work on solving many problems at the same time.

A telephone system is distributed because it can simultaneously connect independent calls, and many unrelated conversations are transmitted over the system at once.

An orchestra is a parallel system because all members of the orchestra are dedicated to producing one outcome. It is similar to a single machine with multiple processors working together on solving a single problem. This would be distributed computing if we had multiple machines, connected with a network, working together on solving a problem in the context of also doing other things. Using all of the system resources, everyone in the orchestra is playing from the same sheet of music and has to work together, in concert, to produce the right sounds [Rewini 98].

Very often applications need more computational power than a sequential computer can provide. One way of overcoming this limitation is to improve the operating speed of processors and other components so that they can offer the power required by computationally intensive applications such as Virtual Environments [Buyya 99a] [Buyya 99b].

A computing *cluster* is a collection of interconnected computers working together. This cluster functions as a single system from the point of view of users and applications. Such clusters can provide a cost-effective way to gain features and benefits that have historically been found only on more expensive proprietary shared memory systems [Buyya 99a]. Shared memory clusters offer a simple and general programming model, but they suffer from scalability problems.

Parallel and distributed programming involves more challenges than serial programming, such as data and task partitioning, task scheduling, and synchronization [Crowcroft 95] [Clark 90] [Shoch 82]. Writing parallel and distributed software may require substantial investment of time and effort [Rewini

98]. Software performance is greatly affected by bandwidth and message latency, both of which are difficult to predict without direct measurement.

A distributed system using message passing may operate in synchronous, or asynchronous modes. In synchronous mode, the program sends a message to a neighbor computer and must listen for the response before it can send another round of messages. Asynchronous mode is very flexible; a program can send multiple messages to neighbor computers before receiving any replies and communication delays are unpredictable. Unlike synchronous mode, the program can continue working and is not forced to wait for each reply [Tagg 97]. (DLove operates in asynchronous mode, making its evaluation more complex.)

Flynn's Taxonomy

A taxonomy is a classification of a large set of items into a smaller number of representative classes. In 1966 M. J. Flynn proposed a taxonomy of parallel computing architectures [Flynn 66]. Parallel systems are classified as having one of two types of instruction streams, single (SI) or multiple (MI) instruction streams, and two types of data streams, single (SD) or multiple (MD) data streams. [Fountains 94] [Dowd 98]

In a Multiple Instruction Multiple Data (MIMD) architecture, each processor has its own set of instructions, which are executed under the control of the control unit associated within that processor. MIMD computers can be further classified into

Multiprocessor Systems, where processors share memory, and Multi-Computers, where computers communicate through message-passing instead of shared memory.

MIMD architectures can be further classified into Single Program Multiple Data (SPMD) architectures. In SPMD architectures all processes share the same executable but they may be working on a different set of data [Sunderam 90]. A difference between SIMD and SPMD is that in SPMD architectures, different instructions can be executed at the same cycle [Hwang 98]. DLoVe utilizes a SPMD message-passing distributed model.

Performance Evaluation

Distributed computing attempts to increase processing speed by computing several tasks on otherwise autonomous computers at the same time [Jeffrey 96][Dowd 98].

To evaluate how well a distributed system behaves one must evaluate its performance, by comparing it against other ways of achieving the same result. In practice, these comparisons are difficult to make.

The most often quoted measure of performance in parallel and distributed systems is *speedup*. The speedup is computed by dividing the time to compute a solution to a certain problem using one processor, by the solution time using N processors. Let S be the speedup achieved by using N processors instead of one processor to solve a problem:

$$S = \frac{\text{One_processor_time}}{N_processor_time} = \frac{T(1)}{T(N)}$$

where $T(1)$ is the processing time of the program when executed on a single processor, and $T(N)$ is the time taken to solve the same problem using N processors. For example, if solving a problem using one processor takes 60 seconds and solving the same problem using N processors takes 20 seconds, then the speedup S is:

$$S = \frac{60}{20} = 3$$

Amdahl's Law [Amdahl 67] [Rewini 98] [Buyya 99a] describes limits of how an application can use parallel processing. Amdahl's law states that the speed of a program in execution on a multiple-processor computer is limited by its slowest sequential part. According to Amdahl, a program contains two types of calculations, those that must be done serially, and those that can be executed in parallel on an arbitrary number of processors. If the time taken to do the serial calculations is some fraction β of the total time τ , $0 < \beta \leq 1$, then the parallelizable portion is $(1-\beta)\tau$ of the total time τ . If we suppose that the parallelizable portion achieves linear speedup, for example, using N number of processors a problem can be solved N time faster, then the speedup on N processors will be:

$$T(1) = 1\tau$$

$$T(N) = \beta\tau + \frac{(1-\beta)\tau}{N}$$

$$S = \frac{T(1)}{T(N)} = \frac{1}{\beta + \frac{(1-\beta)}{N}} = \frac{N}{\beta N + (1-\beta)}$$

The serial part of the program can be computed in time equal to $\beta\tau$ and the parallel part of the program in time $(1-\beta)\tau/N$ because the ideal case where “N workers can do the job in $1/N$ of the time of one worker” [Kenneth 97].

For example, assume that a program consist of 35% ($\beta = 0.35$) code that cannot be executed in parallel, and 65% of code that can be executed in parallel. The speedup of using 10 processors is:

$$S = \frac{10}{0.35 * 10 + (1 - 0.35)} = 2.41$$

This tells us that by using 10 processors we can solve the same problem 2.41 time faster.

Later Amdahl's law was challenged by John Gustafson and Ed Barsis, who showed that for some problems the regularity of the problem can feed as many parallel processors as are needed. Therefore, by adding processors, the size of matrix calculations can grow without bound [Rewini 98] [Kenneth 97].

Gustafson and Barsis [Gustafson 88b] show that problem size s and N are not independent of each other. Gustafson-Barsis law is relative to Amdahl's law but with an assumption about the problem size. Gustafson-Barsis law says that the size of the problem and the number of processors increases together, thus by adding more processors the speedup can increase accordingly. In Gustafson-Barsis law, $T(1)$ is equal by the amount of time needed to compute the sequential part of the program plus the parallel part that can be executed on N processors:

$$T(1) = \alpha\tau_{(s)} + (1-\alpha)\tau_{(s)}N$$

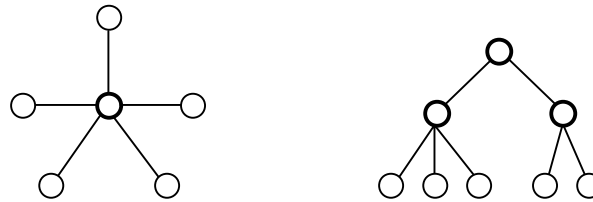
$$T(N) = \alpha\tau_{(s)} + (1-\alpha)\tau_{(s)}$$

Substituting into the speedup equation yields:

$$S = \frac{\alpha\tau_{(s)} + (1-\alpha)\tau_{(s)}N}{\alpha\tau_{(s)} + (1-\alpha)\tau_{(s)}} = \frac{\alpha + (1-\alpha)N}{\alpha + (1-\alpha)} = \alpha + (1-\alpha)N$$

Parallel software

Parallel Virtual Machine PVM is a message-passing software system that allows the utilization of a heterogeneous network of parallel and serial computers as a single computational resource [Sunderam 90]. PVM provides an emulating SPMD architecture. It consists of a library of routines for initiation and termination of tasks, synchronization, and altering the virtual machine configuration. A PVM application is made up of a number of separate sequential programs that cooperate to jointly provide a solution to a single problem. Each program corresponds to one or more processes in a parallel execution. PVM parallel applications can utilize many different communication patterns. One of the most common patterns is the star graph. In the star graph, the middle node is called the supervisor or coordinator and the rest of the nodes are workers. Another form of PVM communication is a tree. The root of the tree is the supervisor, and underneath there are several levels of sub-supervisors, with workers at the leaves.



Star graph (left), and tree graph (right)

The biggest advantage of PVM is its flexibility that includes portability, interoperability between heterogeneous platforms, and fault tolerance. Not only can a PVM program be executed on different platforms running the same operating system, but it can also be executed on an environment that consists of multiple different platforms running different operating systems. This makes it very amenable to its use as a parallel programming tool in typical clusters, which consist of heterogeneous platforms. PVM provides other various run time features such as dynamic spawning of tasks, dynamic changes to the cluster on which a PVM program is being run, and dynamic process groups. These allow a PVM application to incorporate fault tolerance and load balancing by detecting changes in the cluster and moving tasks from one machine to another in their cluster in response to such changes.

PVM's performance suffers because of the flexibility its framework supports, including dynamic task management, load balancing, and heterogeneity. Its set of library functions supporting point-to-point communication is not as rich as that for MPI. For example, PVM does not support the truly asynchronous receive of a message sent from one task to another, as MPI does using message buffering [Clark 90].

The Message-Passing Interface (MPI) is a standard specification designed for writing distributed memory parallel processing utilizing message-passing. Like PVM, it provides library routines that can be called from C and FORTRAN programs and can utilize star and tree graphs for communication. MPI provides a rich collection of point-to-point communication routines and collective operations for data movement, global computation, and synchronization [MPI 94] [Pacheco 97a]. MPI attempts to establish a practical, portable, efficient, and flexible standard for message-passing [Buyya 99a] [Buyya 99b].

The biggest advantages of MPI over PVM are its performance, a larger collection of point-to-point communication models, support for hardware-provided multicast and broadcast, and a larger set of collective communication calls.

However, MPI is not as flexible as PVM in that it does not support dynamically changing the cluster or creating dynamic groups of processes. In general MPI implementation cannot be run on a heterogeneous cluster consisting of machines of different types, since interoperability is not a requirement of the MPI standard. It neither defines details of a parallel programming environment, such as allocation of tasks to processors. These are left to individual implementations of the standard, and thus results in diverse ways to accomplish the same goal, leading to a further variation between the MPI environment across platforms.

DLoVe and other distributed systems

There are three main differences between DLoVe and other parallel systems. While DLoVe's tasks appear externally similar to those in PVM, task allocation is done at compile time, so that there is not appreciable overhead for task management. The purpose of task allocation, in DLoVe, is to allow the Coordinator to always request the same Variables from the same Workers. In other words, the queries the Coordinator sends to the Workers are partitioned, so that the Workers can execute multiple different queries in parallel.

DLoVe's task handling, unlike PVM or MPI, is designed to support multi-user, multi-input application development. Adding a second user to DLoVe's framework, adds a second Coordinator. This means that the Workers now have to serve requests for both Coordinators making each of the Workers work harder, consume more resources, and load the network with more messages.

The third difference concerns performance requirements. DLoVe is designed primarily for Virtual Reality applications and thus requires high frame rate. Distributed applications using DLoVe's framework are characterized by real-time computations and constraints. Thus, not only number of evaluations, but also timing factors need to be taken into account when evaluating DLoVe [Jeffrey 96].

Timing constraints in DLoVe arise from interaction requirements between the Coordinator and the user, and between the Coordinator and the Workers. The communication between the Coordinator and the Workers is described by three operations: sampling, processing, and responding. The Coordinator continuously samples data from the input devices. Sampled data is sent to the Workers that

process it immediately. Then the Workers send the processed data back to the Coordinator in response to its request. All three operations must be performed within specified times; these are the timing constraints [Jeffrey 96] [Bran 94]. For example, if the user moves his real hand, and the movement of his virtual hand appears after a couple of seconds on the screen, the user may be confused and disoriented, making the application unusable and definitely not a Virtual Reality application.

Chapter 5: Related Work - Multi-user UIMS/CSCW Software for Virtual Reality

Introduction

The recent explosion in the quantity and quality of user interface development environments is simplifying or even eliminating the need of programming the Graphical User Interface (GUI) of an application. However, current tools address only single-user/single-machine applications. Distributed, multi-user applications are much more complex. Pieces of such applications typically run in separate address spaces, often on heterogeneous networks of machines. The pieces must communicate and synchronize with each other, sharing and replicating data as needed, and must handle users' interaction at each site.

Diamond Park was designed at the Mitsubishi Electric Research Laboratory (MERL) using SPLINE (Scalable Platform for Large Interactive Environments). Diamond Park is a social virtual reality system in which multiple geographically separate users can speak to each other and participate in joint activities in a mile-square virtual prototype. In this park human visitors can interact with each other and with computer simulated tour buses and autonomous animated figures [Waters 96] [Anderson 95].

Much software has been developed and many techniques have been tested; each has its advantages and disadvantages. Some systems were designed for a specific application domain, such as DIS and SimNet, for military simulations. Other systems, such as RENDEZVOUS and Visual Obliq, described in chapter 3, are not designed for VR systems and thus they do not need to handle the VR necessities described in chapter 2.

RENDEZVOUS is a language and architecture to help people build interactive multi-user systems using constraints and callbacks [Hill 92] [Hill 94]. Visual Obliq [Bharat 94] is a user interface development environment for constructing distributed, multi-user applications using callbacks, and distributed callbacks [Bharat 94].

Systems that use Dead Reckoning

SimNet, a research simulation system, was developed in the early 1980's the DARPA, Defense Advanced Research Projects Agency, and the US Army to

demonstrate the feasibility and effectiveness of networked training. Currently, over 200 SimNet tank trainers are in use at four locations throughout the world. To achieve this massively distributed environment, SimNet proposed a new concept based on the old idea of "dead reckoning". Dead Reckoning means that the current location of any object (such as a tank) can be extrapolated from its previous position and velocity. It works as follows: during any one simulation loop, a tank simulator computes its next position and orientation based on a dynamic model that takes into account a number of factors, including previous position and velocity, terrain grade, engine speed, and soil conditions. It then computes its position again, this time using a limited set of inputs. Next, using the same set of limited inputs, the tank simulation extrapolates the position of all other remote tanks in the simulation. If the difference between the first detailed calculation and the second limited state calculation exceeds a certain threshold, then the tank broadcasts a network message containing its updated position information based on detailed calculation. If the difference between the two calculations is below the threshold, the tank simulation does not generate a new network message. This way, the vehicles send messages only when needed to maintain simulation accuracy. Because all the tank simulators follow this procedure, it drastically reduces the number of network messages. [Johnston 92]

An additional advantage of the Dead Reckoning approach is the ability of the simulation network to recover from breakdown. If one or more units leave the network for a short period of time, the other units will not notice the visual miscues because of the repeated extrapolations. When a unit comes back on line, its true position is smoothly integrated. The benefits of dead reckoning do not come without a price, and in this case, positional accuracy is the tradeoff. Because an object's

position and orientation are extrapolated rather than exactly calculated, objects may appear in different locations on different units.

Although Dead Reckoning works well in DIS, it is not suitable for general-purpose VR systems where the behavior of the objects and the users in a Virtual Environment is unpredictable. Dead Reckoning works well in DIS because the position of every object can be extrapolated from its previous location.

The History-based approach also uses Dead Reckoning but in a more efficient way to get better results than DIS [Singhal 95] [Singhal 94]. The history-based approach offers smooth, accurate visualizations of remote objects, which is providing a scalable solution.

Dead reckoning techniques are central to the large virtual environments targeted by the Distributed Interactive Simulation (DIS) protocols. The current DIS protocol transmits position, velocity, and acceleration information whenever the remote object model exceeds a threshold or a five second timeout elapses. Using the most recent position, velocity, and acceleration information, DIS dead reckoning algorithms generate a second-order model to predict the future object location. The position history-based protocol performs at least as well as DIS for smooth object motion, and it potentially performs better than DIS for non-smooth object motion while requiring less network bandwidth.

The history-based protocol actually performs comparably to the DIS protocol in networked applications containing smoothly moving objects. The effectiveness of the DIS protocol is limited because the tracking relies on acceleration information. An object's acceleration can change more rapidly than its position, so if packets are

delayed, then DIS algorithm is likely to use out-of-date information to predict object behavior. The position history-based protocol is potentially superior to the DIS protocol for tracking non-smooth object motion.

The DIS protocol is highly sensitive to sudden acceleration changes because the algorithm utilizes only the most recent update information. Better performance on these non-smooth paths makes the history-based protocol more useful than DIS protocols in virtual reality applications and visual simulations where entities move in unpredictable ways. [Singhal 95] [Singhal 94]

Another software that implements Dead Reckoning is the Log-Based Receiver-Reliable Multicast (LBRM) [Holbrook 95]. The LBRM protocol provides scalable, timely dissemination of state updates, meeting the needs of multicast sources like DIS terrain entities. The application chooses a threshold according to the freshness requirement of the data being disseminated. Shortening the threshold results in fresher data, but more network traffic. For entities with strict real-time delivery requirements, the threshold must be small.

In the Log-Based Receiver-reliable Multicast (LBRM) approach, reliability is provided by a logging server that logs all transmitted packets from the source. When a receiver detects a lost packet, it requests the missing packet from the logging server. The logging server needs not be co-located with the source host, but if the two are separated, then the source must retain the data until it has received a positive acknowledgment from the logging server. The logging server may be replicated to provide greater reliability (distributed logging server, one primary and multiple slaves where the primary talks to the slaves). The use of a logging server for reliability generalizes the buffering of outstanding data performed by the sender in a

conventional transport protocol [Holbrook 95]. In TCP, the buffered data at the sender is effectively a log of transmissions, from which acknowledged packets have been flushed.

Update Filtering and Area Of Interest (AOI)

Update Filtering is a partition of a Virtual Environment into a large number of “cells”, much like the way the cellular telephone system works. Each host participating in the simulation determines an Area of Interest (AOI), consisting of a number of cells within its range of vision [Macedonia 95a] [Macedonia 95b] [Macedonia 95c].

The DIS system uses multicasting, and that approach seems promising for more general distributed VR application as well. The idea is that any given host, in addition to having its own Internet address, can belong to a number of “multicast groups”. Each multicast group has its own special Internet address, which belongs to a certain range of addresses that have been set aside for this purpose. When any host sends a message to a multicast address, that message is sent to all the hosts that belong to that multicast group. In effect, it is like broadcasting to a subnet that spans continents. Sending a multicast message is better than having to send point-to-point messages to every host in the simulation, so it is easier on the sender.

The most important problem with multicasting is that it is not universally implemented. Some systems have it and some do not. An effort is underway to link small pockets of multicast-capable machines to each other over the Internet; the

result is a “multicast backbone” or MBONE. The MBONE system uses “tunneling”; it wraps the IP packets destined for a multicast address inside another IP packet, which travels through the regular Internet from one MBONE subnet to another [Casner 94]. How does multicasting fit in with Area Of Interest management? In the DIS system, each cell has its own multicast group address; in other words, there is a one-to-one correspondence between cells and multicast addresses. As participants move around, they enter and leave cells; this corresponds to entering and leaving multicast groups. Since participants only receive messages for multicast groups they are in, the multicast system itself implements the Update Filtering.

Key Frame Animation

Using the Key Frame Animation approach, a designer can design the geometry of objects and also specify their behavior. Key Frame Animation can be used in Virtual Environments where the position and the behavior of the users or simulated objects is unpredictable.

JDCAD+ is an interactive 3D geometry modeling system and animation editor [Halliday 94]. These objects can be imported in a Virtual Environment and users can interact with them. Behaviors such as walking, jumping, and drinking from a cup are very hard to code by hand because of the complexity of the motion. JDCAD+ uses key frame animation, which gives the ability to the programmer/designer to specify key frames of the motion of the objects so that the system will know what the next key frame is and animate the motion. After editing

the key frame animation, JDCAD+ produces OML code for the behavior. This code can be edited/modified later if someone wants to alter the behavior of the object.

In systems such as The Alias System [Alias] the animation sequences are not interactive. Once a keyframe sequence starts there is no way of interrupting it, and thus the object cannot respond to events that occur while a keyframe sequence is running. In JDCAD+ however, the keyframe process is implemented by OML code that can respond to events occurring in the environment. In The Alias System, once a keyframe sequence has been defined there is no way of modifying it or combining it with other types of motion.

Other User Interfaces

Repo-3D is a general-purpose, object-oriented library for developing distributed interactive 3D graphics applications across a range of heterogeneous workstations. It emphasizes developing applications for Computer Supported Cooperative Work (CSCW) and Distributed Virtual Environments (DVEs). All shared data is fully networked transparent because it is encapsulated within the programming language objects. Distribution of new objects between the processes is as simple as passing them back and forth as parameters to, or return values from, method calls. The underlying system takes care of the rest with no additional effort on the part of the programmer. In addition, updates to objects are automatically reflected in all replicas, the same way DLoVe works.

There are times however, when a shared graphical scene may need to be modified locally (Local Variation); this is supported by Repo-3D and in limited form in DLoVe. For example, a programmer may want to highlight the object under one user's mouse pointer without affecting the scene graph viewed by other users.

Most high-level graphics libraries, such as Inventor [Strauss 92a] and Java 3D [Sowizral 98], do not provide any support for distribution. Others, such as Performer [Rohlf 94] [Zyda 93], provide support for distributing components of the 3D graphics system across multiple processors, but do not support distribution across multiple machines.

TBAG [Elliott 94], a high-level constrained-based, declarative 3D graphics framework, scenes are defined using constrained relationships between time-varying functions. TBAG allows a set of processes to share a single, replicated constrained graph. This is done by two sets of functions. The "externalization" functions map constrainables and assertions into machine-independent identifiers that can be passed to different processes of different machines. The other set of functions "internalizes" the externalized identifiers back into C++ constrainables and assertions. The programmer thinks of the externalize/internalize sequence of calls as providing access to existing constrainables or assertions in a separate process, thus allowing constrainables to be asserted on and values to be retrieved from remote constrainables. Any changes made to any constrainable are reflected in all processes that access that constrainable. Whenever a process creates a constrainable, all other involved TBAG processes create "clones" of that constrainable. Then, whenever a constraint is asserted/retracted in any process, all related TBAG processes are informed of that assertion/retraction and perform it themselves locally. Thus, each process has a semantically identical copy of the

entire constraint network, which makes this TBAG limited in scalability because all processes have a copy of (and must evaluate) all constraints, whether or not they are interested in them. DLoVe works the same way but it only evaluates the constraints that it needs, not all of them. TBAG, unlike Repo-3D and DLoVe, does not support local variations of the scene in different processes. [MacIntyre 98]

Workroom is a general-purpose simulation infrastructure that allows multiple participants to interact in unpredictable ways while minimizing simulation latency. Client processes manage the local simulation, interact with mechanical devices such as motion bases and position trackers, and perform image generation. In some cases, a separate, dedicated client process handles interaction with devices such as the MR toolkit [Shaw 93]. The server process has the responsibility of maintaining the state of all simulation objects running on all the clients. The server handles all requests for data about the environment or the state of each simulation object.

For example, if there are two clients and each is simulating the behavior of a differently flying colored object, the server keeps track of each flying object's location. When the server is queried, it sends a data packet containing the location, velocity, and color for each flying object.

In Workroom, and in DLoVe, dead reckoning cannot be implemented because of the unpredictable participants' behavior (humans unlike vehicles or projectile are not able to extrapolate their positions). The way to control objects in these systems is by sending network messages. And because there are too many messages on the network, the network may be overloaded. One solution to this problem, which is implemented in DLoVe, is to use low and high priority messages. The system can lose low-priority messages (like current object position) without consequence. High-

priority messages (like constraint attachment) use confirmation or redundancy schemes to ensure that a message arrived at its destination [Pimentel 94].

NPSNET [Macedonia 95a] [Macedonia 95b] [Macedonia 95c] [Macedonia 97], uses multicasts, instead of broadcasts like SimNet, and instead of point-to-point communication like DIVE (discussed below) [Carlsson 93] and MASSIVE which supports up to only ten users [Greenhalgh 95a] [Greenhalgh 95b]. The MR Toolkit [Shaw 93] uses an internal client/server model for communicating between I/O devices and the application. However, point-to-point UDP communication is used to maintain consistency between users. In addition, it uses an example where cells are hexagonal, somewhat like a strategy board game or certain types of military simulation computer games. This hex grid is well suited to military simulations, and is a closer model for circular Areas of Interest than a square grid would be. As a participant moves around, cells will enter and leave their Area of Interest; at any given time, they are only receiving updates for cells they can see, resulting in a small, manageable number of updates. The combination of AIO filtering and dead reckoning produces significant bandwidth savings.

The Performance Architecture for Advanced Distributed Interactive Simulation Environments (PARADISE) project [PARADISE] [Singhal 99] in addition to focusing on the graphical aspects of networked VR design, it also addresses network software architecture issues that environments containing thousands of users, face. The PARADISE system used IP multicast, assigning a different multicast address to each active object in the VE. However, because the early workstations available to the group did not support multicast at the time, they implemented a multicast simulator on the local network. PARADISE uses a similar mechanism to transmit updates for local objects in much the same way as SIMNET and DIS. To further

reduce bandwidth, a hierarchy of area of interest (AOI) servers is used to collect subscriptions from each host. The servers monitor the positions of objects and notify hosts to which objects' multicast groups they should subscribe.

Unlike SIMNET, PARADISE treats all objects uniformly as first-class entities. In addition, PARADISE recognizes that in a VR environment there are some objects that change their positions rapidly and some others slowly. This means that the rapidly changing objects need to send updates more frequently than the slowly changing objects.

To support rapidly changing entities, PARADISE uses improved dead reckoning protocols such as Position History-Based Dead Reckoning (PHBDR) [Singhal 95], which transmits smaller update packets and provides better accuracy when objects move wildly [Singhal 96]. To support slowly changing entities, PARADISE focused on reliable multicast protocols to eliminate the frequent heartbeat messages present in DIS. Log-Based Receiver-Reliable Multicast [Holbrook 95] provides a lightweight reliable multicast service that includes a persistence mechanism.

The Swedish Institute of Computer Science Distributed Interactive Virtual Environment (DIVE) is another early and ongoing academic virtual environment [Carlsson 93] [DIVE] [Hagsand 96]. DIVE uses a distributed, fully replicated database similar to SIMNET and DIS-compliant systems. Unlike SIMNET and DIS, DIVE's entire database is dynamic. DIVE has the capability to add and remove new objects and modify the existing databases in a reliable and consistent manner. This is implemented via a reliable multicast protocol for distributed database locking, adding significant communication overhead. And because of this DIVE is difficult to scale beyond 1 to 32 participants. However, DIVE does well in situations where

database changes must be guaranteed and accuracy is a major criterion such as in collaborative environments.

Chapter 6: Basics of DLoVe

Introduction

DLoVe is designed on a two-component model for describing and programming the fine-grained aspects of non-WIMP (non- Window Icon Mouse Pointer) interaction (such as virtual environments). It is based on the notion that the essence of a non-WIMP dialogue is a set of continuous relationships, most of which are temporary. This model combines a data-flow or constraint-like component for the continuous relationships with an event-based component for the discrete interactions, which can enable or disable individual continuous relationships.

Other current Graphical User Interfaces (GUIs) or (WIMP) interfaces, also involve parallel, continuous interaction with the user. But, most other user interface description languages and software systems are based on serial, discrete, token-based interaction.

DLoVe is designed to provide a fundamentally continuous, rather than discrete, treatment of naturally continuous phenomena such as time and motion. However, it does treat discrete events as discrete events and it provides a mechanism for communication between the continuous and the discrete sub-system.

Continuous Time

DLoVe's sub-system consists of object elements that define the relationship between variables. The entire set of these elements connected together form a constraint-like graph. Changes on one end of the graph propagate to the other end. Interaction that is conceptually continuous is encoded directly into these elements, and thus the application does not need to deal with tracking events from conceptually continuous devices. Examples of conceptually continuous interaction include, drinking from a cup in a virtual world, throwing a ball in a virtual park, and driving a car in a virtual city.

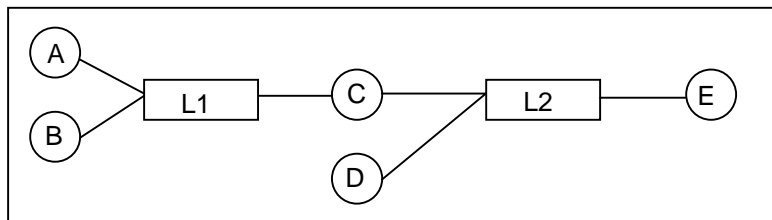
Continuous time in DLoVe is handled via Variables and Links, explained below. Based on these Links and Variables a network can be designed to describe the behavior of objects and interaction techniques.

Variables are objects in DLoVe that store values and know which Links need them as inputs and which for output. They are invariant data flow graph elements that serve as both continuous and short-term data repositories. Some Variables are directly connected to input devices, some to outputs and some to application

semantics. They are used for communication within the user interface model or they are just used to hold intermediate results of Link calculations.

Links are objects that contain functions and are attached at both ends to Variables. Links get input from Variables and place the result of their calculations into other Variables. The body of a Link specifies how the attached Variables are related. Links can be enabled or disabled in response to user inputs. When a Link is disabled, it is as if this Link were not part of the constraint network anymore. By enabling and disabling Links we can quickly change the constraint network on the fly since only a flag needs to be set or cleared to indicate that a Link is enabled or disabled.

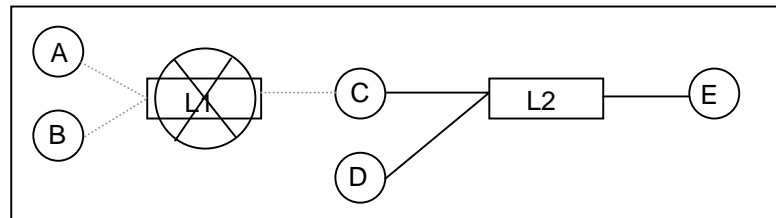
Conditions are also provided to enable and disable groups of Links instead of enabling or disabling Links individually. For example, we can attach five Links to a Condition; every time we want to enable all five Links we can just enable this particular Condition, which then enables all five Links individually.



Links and Variables

The above diagram shows the Variables as circles and the Links as rectangles. When a Link is created, it is enabled by default. In the DLoVe constraint graph I

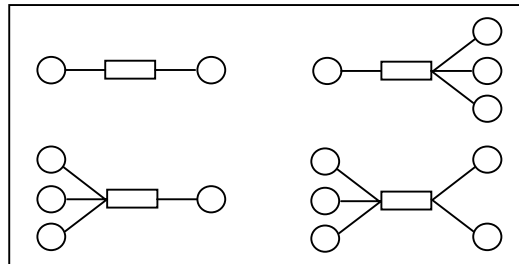
draw a crossed circle on top of a Link to indicate that is disabled. A DLoVe graph is read from left to right. For example, the Variables 'A' and 'B' are inputs to Link L1, and the Variable 'C' is its output Variable. When L1 is disabled a crossed circle is drawn on top of it and it is as if this Link was deleted from the network. However, a disabled Link is still part of the data structure and when it becomes enabled again it knows how it is supposed to be attached to its Variables. In this case, if the Link 'L1' is disabled, the graph would look like the following figure.



A disabled Link

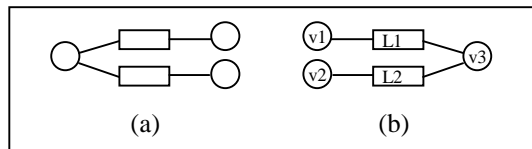
The relationship between 'A', 'B' and 'C' is terminated temporarily until 'L1' becomes enabled again.

Links can have multiple outputs, unlike DeltaBlue [Freeman-Benson 90], and multiple inputs. For example, all of the following combinations are valid in a DLoVe constraint network.



Combinations of Variables attached to a Link

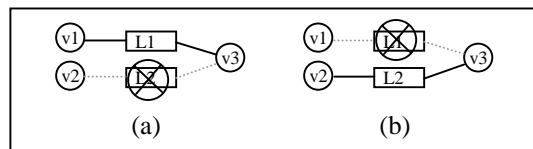
In addition, a single Variable may be used as input or output to multiple Links, as illustrated below.



Combinations of Links attached to Variables

When a Variable is used as output to more than one Link, the result may be unpredictable, since both Links try to bring the Variable up-to-date using different constraints, as in (b). In this case, Link L1 and Link L2 try to bring Variable v3 up-to-date using different dependencies - L1 uses v1 and L2 uses v2 -. DLoVe will use the Link that was created last. For example, if L2 was created after L1, DLoVe will update Variable v3 according to the constraint in Link L2. This might seem confusing and ambiguous, but it is very useful when only one of the two Links is enabled at a time.

Let us say that in a virtual factory there are moving objects on an assembly line. Let us also assume that Variable v2 is time and Variable v1 is the position of a virtual hand. When the object on the assembly line is moving, only Link L2 is enabled. This way, the object's position depends on time. However, if the user grabs the object with his virtual hand to examine the moving part/object, Link L2 becomes disabled, while Link L1 becomes enabled. Now the user has the ability to lift the object off the assembly line and examine it. The object is no longer on the assembly line, and its position does not depend on time. The constraint graph in the following figure is (a) when the user examines the object, and is (b) when the object is moving on the assembly line.



A Variable used for output by multiple Links

Discrete Time

There are however, other interactions that are fundamentally discrete (event-based) and for that I use the event-based component of DLoVe. Such examples include button presses, menu choices and gesture recognition verifications. TBAG applications [Elliott 94] generally deal with such discrete input events by retracting some existing constraints and asserting new ones. Bramble uses a similar mechanism [Gleicher 93].

DLoVe handles the discrete time using Event Handlers, objects that capture tokens and respond to them. Event handlers contain a user specified body that describes the response to tokens. The application sends a token to all event handlers, and only those event handlers that are interested in the token execute their body. The responses might include setting Variables, making custom procedure calls and setting or clearing Conditions on Links. Event handlers recognize states and state transitions, and can provide different services depending on the state they are in. For example, the user might intersect his/her virtual hand with a virtual object. The event handler will receive an INTERSECT token and it will move to its 'intersect' state. In this state if the user presses the left mouse button, the event handler enables a Condition and transitions to the 'dragging' state. As a result the object is now attached to the virtual hand so that wherever the user moves his/her hand the object follows the movement of the virtual hand. When the user releases the mouse button, the event handler disables the Condition, transitions to the 'start' state, and the continuous relationship hand-object is terminated.

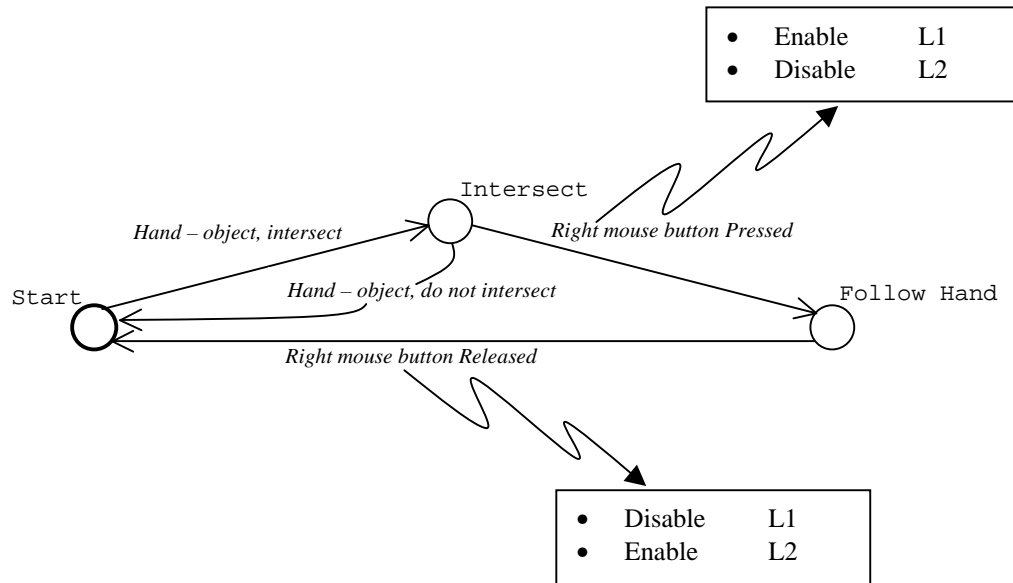
A token is a structure similar to a record in Pascal or struct in C. It contains a timestamp, an id, and other optional fields such as position and other user defined variables. For instance, when the user's virtual hand intersects with a virtual object, the user can press the left mouse button to send token. This token indicates that the virtual object should be attached to the user's hand position in space, so that when the user moves his/her hand, the object moves along with his/her hand. Releasing the mouse button sends another token that indicates that this hand-object relation should be terminated. In this case, when the user's hand intersects with the virtual object and then the user presses the left button; the LEFTDN token is send to all event handlers. The event handler that is responsible for attaching the

virtual object to the user's virtual hand enables the appropriate Link so that a hand-object relation is established.

The application reads all X events and hands them over to a DeviceXWindow DLoVe object. This object then turns X events into tokens and sends them to all event handlers.

Enabling a Link is similar to asserting a constraint, and disabling a Link is similar to retracting a constraint from the constraint graph. Even though enabling and disabling Links is similar to retracting and asserting constraints, enabling and disabling occurs more quickly; Enable only marks a Link as part of the graph and Disable as not. The global structure of the constraint graph does not vary. When a Link becomes enabled, a single flag is cleared to indicate that this particular Link is part of the constraint graph again. There is no need to remember and set all the dependencies, since the object was never deleted from the memory. This makes DLoVe very efficient for modifying the constraint graph on the fly. At creation time the programmer specifies the dependencies of a Link and does not have to remember them ever again when he/she needs to assert/retract a constraint.

In the example of the factory and the moving objects on the assembly line the Event Handler's state diagram might look like the following:



State Diagram of the Event Handler in the factory example

When the user's hand intersects with the moving object, the event handler receives a token (e.g. "ENTER"), the object becomes highlighted, and the event handler transitions to the 'Intersect' state. At this state, if the user presses the right mouse button, the event handler receives another token (e.g. "LEFTDN"), transitions to the "Follow Hand" state, enables Link L1, and disables Link L2. Now a Hand - object relation has been established, and the object follows the movement of the hand. When the mouse button is released, the event handler receives another token (e.g. "LEFTUP"), transitions to the 'start' state, disables Link L1, and enables Link L2. At this point, the hand-object relationship is terminated.

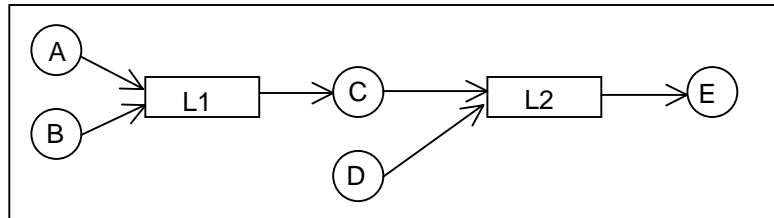
Communication Between Continuous and Discrete Time

The Continuous and Discrete models can communicate with each other to provide a more powerful environment. Event Handlers can enable and disable Conditions/Links and thus re-write the network graph. In other cases, a Link may be reading input Variables and by performing a complex function on them over time it may generate a token that is processed by an Event Handler. This can be useful for gesture recognition or for eye tracking applications where when the user looks at an object for over five seconds, the object becomes selected. An application is presented in chapter 11 where the user selects with his/her eye an object to manipulate.

Data Structures

The network consisting of all the Links and Variables forms a data-flow-like network, where Variables store values and Links satisfy relationships among Variables. DLoVe includes a one-way constraint engine whose primary responsibility is to keep all constraints satisfied when possible. Each Link has an Evaluate() member function which is used to bring its output Variables up-to-date. When the constraint engine needs to bring a Variable up-to-date, it executes the Evaluate() functions of the Links involved in the constraint. This function gives the relation between input and output Variables and is specified by the programmer.

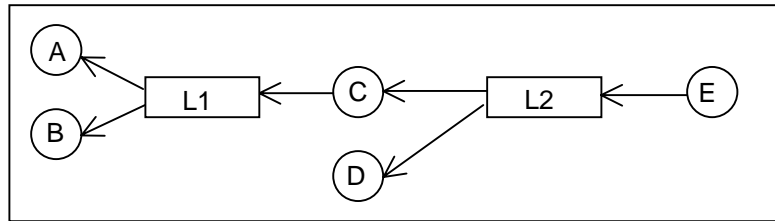
The constraint solver uses an incremental approach that guarantees that Links will not be Evaluated more than once in each iteration and only when needed. The programmer specifies the input and output Variables that each Link uses and the relationship between them, as well as the Evaluate() member function.



Data-flow Graph

In the above figure, incoming arrows to Links point from Variables used as inputs, and outgoing arrows from Links point to Variables used as outputs. The user specifies this along with the Evaluate() member for each Link. The Evaluate() member is a function that describes how the input and output Variables are related with each other. This graph is called the “data-flow” graph because it indicates which Variables produce which Variables. For example, Variables ‘A’ and ‘B’ produce ‘C’, and Variables ‘C’ and ‘D’ produce ‘E’.

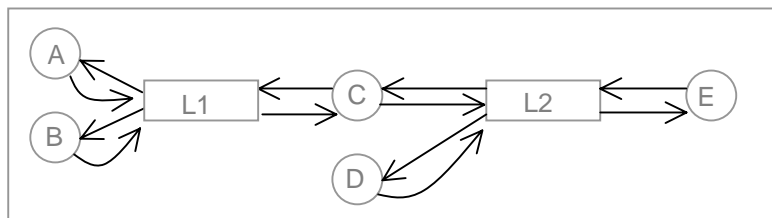
After all Links and Variables are created, DLoVe automatically generates the second set of pointers as shown below:



Data-dependency Graph

In the above graph, outgoing arrows from Links point to Variables that are needed to produce its output Variables. Incoming arrows to Links point from Variables where the result of the execution of the Evaluate() function will be stored. This graph is also called “data-dependency” graph because it indicates which Variables depend upon which other Variables. For example, Variable ‘E’ depends upon ‘C’ and ‘D’. And Variable ‘C’ depends upon ‘A’ and ‘B’.

Using the second set of pointers, the constraint engine can work very quickly to determine incrementally which Variables are out-of-date and which Links need to be evaluated to bring the demanded Variables up-to-date. The constraint graph at the end looks as follows:

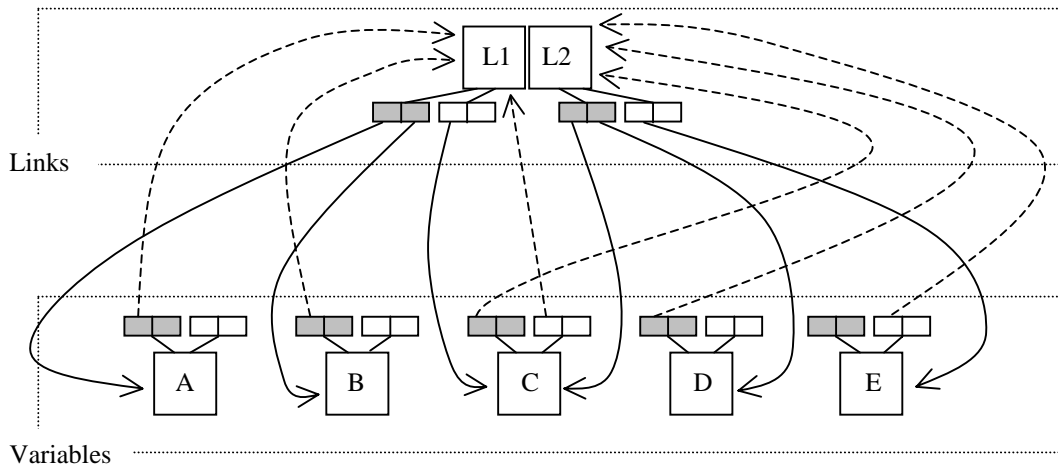






DLoVe constraint Graph

Internally, each Link has two sets of pointers. The first set of pointers points to Variables that the current Link is using as inputs. The second set of pointers points to Variables that are used as output by the current Link.

Just as each Link has two sets of pointers, each Variable has two sets of pointers. The first set of pointers point to Links that use the current Variable as input, where the second set of pointers points to Links that use the current Variable as output.

The data structure of the above diagram is shown below:



- Links
-  Pointers to Input Variables
 -  Pointers to Output Variables
- Variables
-  Pointers to Links that use this Variable as Input
 -  Pointers to Links that use this Variable as Output

Data structure that holds the Links and Variables

At the top of the figure are all the Links, each having two sets of pointers. At the bottom are all the Variables, with their two sets of pointers. The programmer sets up only half of the pointers; pointers from Links to Variables. DLoVe sets up the pointers from the Variables to Links automatically.

Internal flags of the Solver

Variables and Links are C++ objects that have member variables and functions. Each Variable contains the following flags:

- dirty
- assigned2worker
- L_counter

The 'dirty' flag indicates that the current value of a Variable has been changed since the last time it was requested. The value of a Variable can be changed by either directly by the user program or indirectly by the constraint solver. When a Variable is out-of-date its 'dirty' flag is set. The constraint solver uses the 'dirty' flag to figure out which Links must be evaluated to bring a demanded Variable up-to-date. Using this flag the algorithm marks which Variables are out-of-date and implements incremental solving by evaluating the Links in topologically sorted order.

The 'assigned2worker' flag is used by the Partition algorithm discussed later. DLoVe uses this variable when it runs in the distributed mode. This variable stores the id of the Worker that is responsible for keeping the current Variable up-to-date.

The 'L_counter' is also used when DLoVe runs in the distributed mode. This variable holds the number of Links that need to be evaluated in the worst case to bring a Variable up-to-date.

Each Link contains the following variables:

- enabled
- visited
- dirty
- mark4partition

The 'enabled' flag indicates whether or not a Link is enabled. A Link that is disabled is not really part of the constraint graph. When a Link is disabled the constraint solver skips the disabled Link as if it was not there. Instead of deleting the object (Link), which specifies the relation of Variables, it only clears the enabled flag to indicate that the current Link is disabled. This speeds up the removal/insertion of constraints.

The 'visited' flag is used to detect cycles in the constraint graph. When a cycle is detected, the iteration stops after it goes through once instead of staying in an infinite loop. When a cycle in the constraint graph is detected, DLoVe evaluates the Links in the cycle once and then it breaks the cycle so that it stops the recursion.

When the 'dirty' flag of a Link is set, the Link must be evaluated even if its input Variables have not changed since the last iteration. We fall in this case only when a Variable is used as input to more than one Link. This is the only case where this flag is used and an example is shown later.

When DLoVe runs in the distributed mode, the Partition algorithm uses the 'mark4partition' flag to partition the graph correctly when there is a Link that uses multiple Variables as input.

Operations on Links

Each Link has an Evaluate() member function specified by the programmer, that tells the relationship between the Link's input and output Variables. When the constraint solver needs to bring a Variable up-to-date, it evaluates all the Links in the path of the Variable, by traversing the "dependency-graph" starting at the Variable in demand. What the solver actually does is execute the Evaluate() function of each Link that needs to be evaluated.

Enabling and Disabling, directly or indirectly via Conditions, are operations performed on Links. Each Link has an Enable() and a Disable() member function that simply set and clear the 'enabled' flag in the Link. As we will see later, because these operations re-wire the graph, when the system runs in distributed mode, these operations must be visible throughout the entire set of machines participating in the distributed environment.

Operations on Variables

Similarly to Links, Variables also have member functions that the programmer can use to operate on their values. The operations that can be performed on each Variable are:

- two flavors of Set (to assign a value to a Variable)
- two flavors of Get (to get a value of a Variable).

The two Set operations are SetI(), which stands for 'Set Internal', and SetE(), which stands for 'Set External'. SetI() is performed within the Evaluate() member of the Link and this is something that the programmer has to know and follow. SetE() is performed by the main program to set a Variable as well. Both Set operations set the value of the Variable and also set the 'dirty' flag of the Variable. Their behavior is identical in the non-distributed mode, but it is very different when the system runs in the distributed mode as it is explained in chapters 7 and 8.

The two Get operations are GetI(), which stands for 'Get Internal', and GetE(), which stands for 'Get External'. GetI() returns the raw value of a Variable and it is mainly used in the Evaluate() member of a Link as the SetI(). If GetI() is called from the main program, it may return the value of the Variable which is out-of-date, because it does not trigger the constraint solver to run. GetE() however, triggers the constraint solver and returns the value of the Variable up-to-date. The behavior of these functions depends on how the system is running. As it is shown later, when the system is running in distributed mode, SetE() and GetE() are sent over the network as requests to Workers from the Coordinators. The return value of a GetE() in the distributed mode is passed back to the Coordinator over the network.

Constraint Solver

To keep the Variables up-to-date, DLoVe implements an incremental constraint solver that supports lazy evaluation, similar to Eval/vite [Hudson 91]. The algorithm finds the set of Links that need to be evaluated to bring the demanded Variable up-to-date. Then, based on topological ordering (using depth-first sorting), it evaluates Links and brings the demanded Variable up-to-date. The solver guarantees that each of these Links will be evaluated only once, after its dependencies become up-to-date.

Each Variable knows if it is up-to-date or not. Using this information the Constraint solver uses an incremental update and evaluates only the Links whose input Variables are dirty or otherwise out-of-date.

Variables may change but that does not trigger the constraint solver to run. The Constraint solver is lazy, which means that it will only update Variables when they are requested by the user or by the constraint solver itself. When a Variable is requested, the `Do_evaluation()` function of the solver is called on the demanded Variable. `Do_evaluation()` clears the 'visited' flag' on all Links, and then calls the constraint solver on this Variable. The `Solve()` function of the constraint solver is the main function that brings Variables up-to-date.

```
1 Do_evaluation(demanded_var)
2   ClearALLvisited();
3   Solve(demanded_var)
```

The Solve() function finds the set of all Links that need to be evaluated to bring the demanded Variable up-to-date, and then each Link is evaluated in order, if needed.

The pseudo-code of the Solve() function is shown below:

```
1  Solve(demanded_var)
2
3      foreach Link link that uses demanded_var as output Variable do
4
5          if link is enabled and not visited then
6              link->SetVisited();
7
8              foreach Variable var that is an input to Link link do
9
10                 Solve(var);
11
12                 if var is dirty then
13                     foreach Link lptr that uses var as input do
14                         lptr->SetDirty();
15
16                 if link is dirty then
17                     foreach Variable vv that Link link is using as input do
18                         vv->ClearDirty();
19
20                 link->Evaluate();
21                 link->ClearDirty();
```

How it Works

Recursively, starting at the Variable in demand, the algorithm traverses the constraint graph and finds all Variables upon which the demanded Variable depends. If at least one of these Variables is dirty (line 12), the algorithm marks all Links that input the current dirty Variable as dirty to indicate that these Links need to be evaluated (lines 13-14). It then clears the dirty flag of all input Variables (lines 17-18), since the current Link's output Variable is up-to-date, and its own dirty flag (line 21), because it was just evaluated. It then calls the Evaluate() (line 20) member function for each such Link to bring the output Variables of the Link up-to-date. Lines 5-6 ensure that only enabled Links are considered. If there is a cycle in the graph, it is broken after detection, so that the algorithm does not fall into an infinite

recursion. DLoVe breaks cycles using the 'visited' flags of the Links, which keep it from going around in cycles.

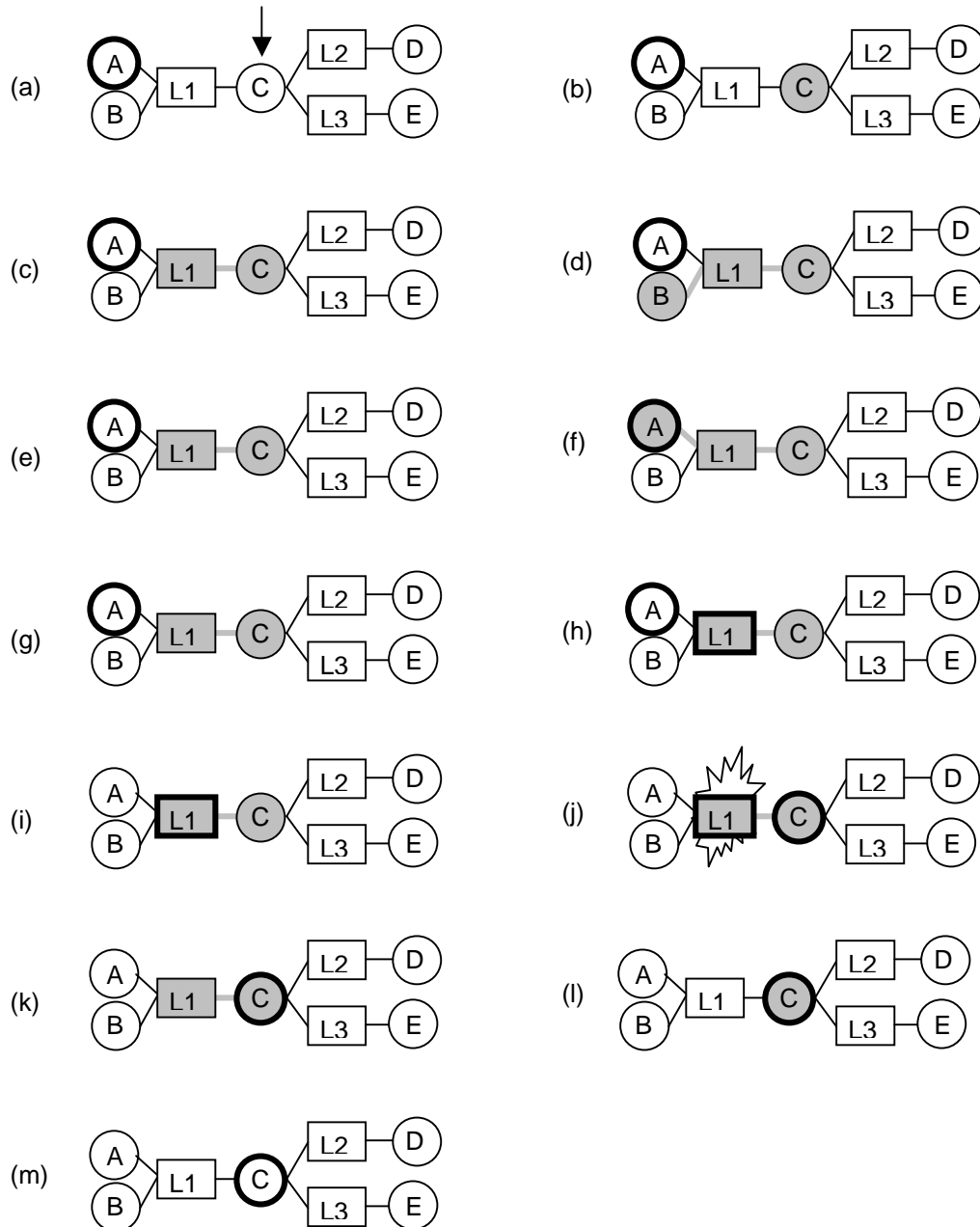
The algorithm is triggered and executed when the programmer calls the GetE() member function of a Variable. The constraint solver is executed and returns the up-to-date value of the demanded Variable that is up-to-date.

All the 'foreach' loops execute very quickly because each Link has stored all the pointers to its input and output Variables. Similarly, Variables have stored all the pointers to Links that need them as inputs or outputs.

The algorithm ensures that first, Links nearest to the changed/dirty Variables are evaluated and so on until the demanded Variable becomes up-to-date. In the following example, the Variable 'A' has changed and the user requests the Variable 'C' as shown in (a). The constraint solver calls Do_evaluation() on Variable 'C' which is the requested Variable (b). Then the algorithm finds Link 'L1', which is the Link that uses 'C' as output Variable (c), (line 3). In (d) the algorithm finds the first input Variable to Link 'L1' (line 8) and calls Do_evaluation() on 'B' (line 10) and returns since there are no Links using Variable 'B' as output (e). Then it repeats on the second input Variable to Link 'L1' which is Variable 'A' and returns (f, g). Lines 12-14 are represented in (h) where the algorithm sets as dirty the current Link 'L1' because at least one of its input Variables is dirty ('A' is marked as dirty).

Then it clears both input Variables 'A' and 'B' (i) (lines 16-18) because it will now bring the output Variable 'C' up-to-date. Line 20 calls the Evaluate() function of Link 'L1' and stores the result in 'C' (j). Now 'C's dirty flag is set because the result in it has just changed. Then (line 21) it clears L1's dirty flag (k) because it was just

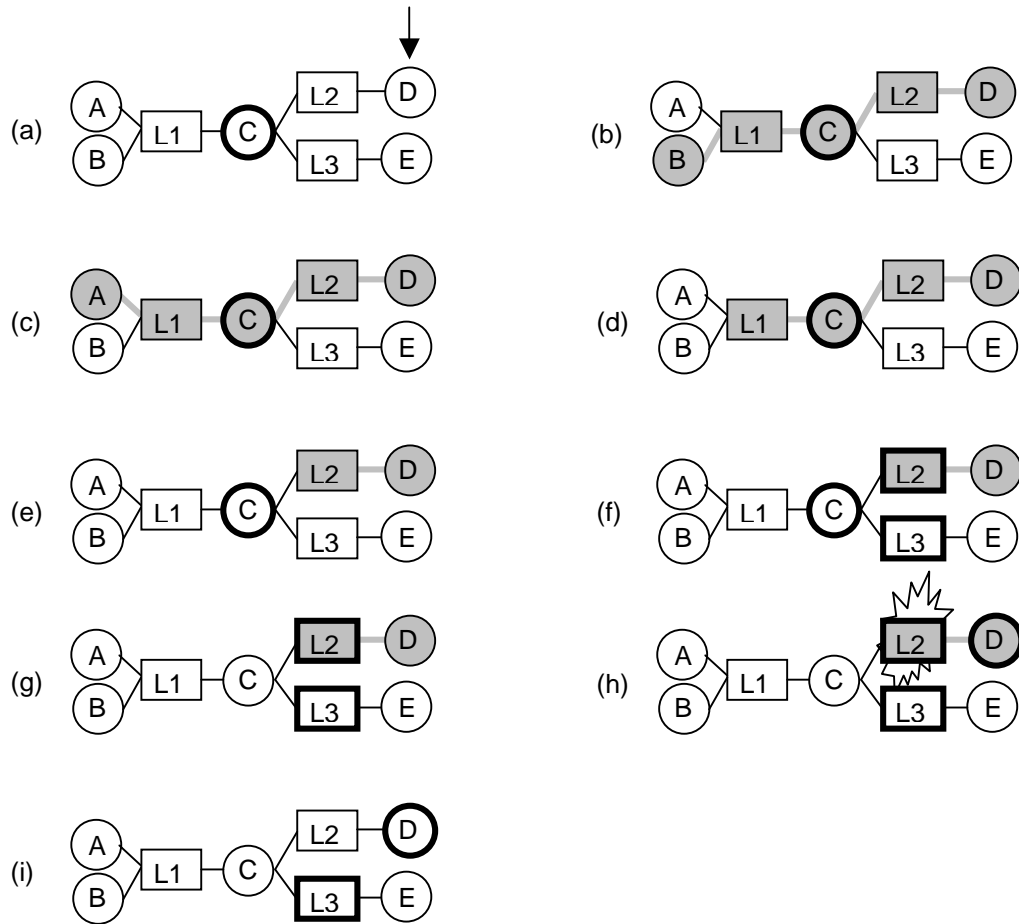
evaluated. Then in (l) and (m) the algorithm unwraps the recursion and finishes, leaving 'C' dirty to indicate that its value has just changed.



Animation of the Solve algorithm

The following example illustrates the incremental behavior of the algorithm. Continuing from the previous example where the value of the Variable 'C' has changed and thus it is marked as dirty, let us say the user requests Variable 'D' as shown in (a). The algorithm then recursively traverses the graph down to Variable 'B' (b) then down to 'A' (c). In (d) (line 16) finds that the Link 'L1' is clean since its input Variables have not changed since last time and unwraps the recursion in (e). In (f) (line 12) Link's 'L2' input Variable is dirty and so it marks all Links that use Variable 'C' as input Variables, as dirty (lines 13-14). This causes both Links 'L2' and 'L3' to be marked as dirty. (g) shows lines 17-18 where L2 input Variables' dirty flag gets cleared. Then the Evaluate() function of L2 gets called (h) (line 20) and the result is stored in L2's output Variable 'D' and so, it is also marked as dirty. Note that the Evaluate() uses SetI() to store a result to a Variable which also sets the 'dirty' flag of that Variable. The algorithm finishes and leaves Variable 'D' up-to-date with its dirty flag set, since it just has been changed, and the 'L3's dirty flag also set.

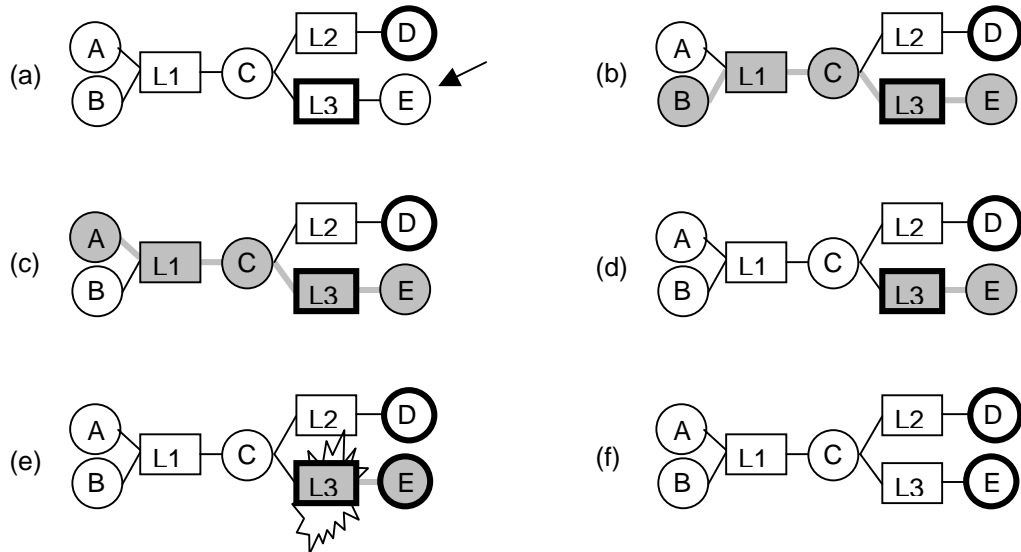
The reason that the algorithm leaves 'L3's dirty flag set is to remember that even though the 'dirty' flag of 'C' is clear, a request on Variable 'E' should trigger the evaluation of Link 'L3' even though the input Variable of 'L3' is not dirty. The reason that the 'dirty' flag of 'C' is clear is because the evaluation of Link 'L2' cleared the 'dirty' flag of 'C'.



A second example of the Solve algorithm

The reason that we need a dirty flag in Links is illustrated in the following example that is a continuation of the previous example. Let us say the user at this point requests Variable 'E' as shown in (a). The algorithm traverses the graph in (b) and (c) and then unwraps as shown in (d) since there were no changes to Variables 'A', 'B' and 'C'. In (d) however, L3's dirty flag is set (line 16) and so, it must be evaluated. Note here that even though L3's input Variable is clean from the previous run, it has to be evaluated to bring the demanding Variable 'E' up-to-date. So, in (e) L3's Evaluate function is called, and the result is stored in its output Variable 'E', that gets marked as dirty since it has just been changed. The algorithm

finished and leaves Variable 'D' dirty, from the previous run, and Variable 'E' as dirty as well from this run.



A third example of the Solve algorithm

Chapter 7: Distributed/Parallel DLoVe

Introduction

One of the major difficulties with existing software lies in transforming a program written for a sequential machine into a program that runs on multiple machines on a network in parallel. A program written in DLoVe however, can be initially written to execute on a single sequential machine; with minor programming effort the same program can execute in parallel on multiple workstations. The only difference between the parallel and serial versions is that different libraries are used in compiling this program. When compiling for parallel execution, compilation generates two different executables, the executable for the Coordinator (the machine mainly responsible for the graphics and the input devices), and another executable for the Workers (the machines responsible for doing constraint calculations). The Coordinator is a workstation with a display device and input devices. It is responsible for reading all data from the input devices and generating the graphics. Workers are only responsible for doing calculations based on the Coordinator's

requests. Workers are workstations that do not need to have any input or output devices attached to them: “headless workstations”.

Basic Structure of the ‘main()’ function

To program in DLoVe, the programmer must construct the program from Links and Variables. This is a similar but simpler process than programming in C++ using inheritance, encapsulation, and overloaded functions. The main program of any application looks like the following:

```
1  main() {
2      Link::InitCommunication()
3      SetUp_Window_System()
4      Create_and_Setup_Links_and_Variables()
5      Link::InitSystem()
6
7      while( true ) {
8          Link::START()
9          read-inputs()
10         request-outputs()
11         draw-display()
12         Link::FSTOP()
13     }
14 }
```

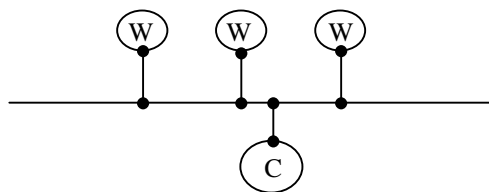
The call to `Link::InitCommunication()` (line 2) connects all Workers to the Coordinator using the DLoVe protocol, built on top of TCP/IP. The program then sets up the windowing system (line 3), and creates all Links and Variables (line 4).

The call to `Link::InitSystem()` (line 5) has different effects on the Coordinator and on the Workers. During this call, the Coordinator partitions the network of Links and Variables, and assigns roles and responsibilities to each Worker. During the same call, the Worker(s) loop forever, reading and satisfying requests from the

Coordinator. The Worker(s) become “servers” listening for requests from the Coordinator and providing services to it.

The rest of the main() function is only executed by the Coordinator. The Coordinator continually (line 7 – 13) reads input devices, requests Variables from Workers (line 10), and renders the display (line 11). The two calls Link:START() (line 8) and Link::FSTOP() (line 12) are used to build multiple requests that are send to Workers in a block message. This is needed to get better performance out of the network. It is better to send fewer, larger messages, than to send many, smaller messages. Later we will see details on the evolution of DLoVe and why I had to pack multiple requests into a single big message.

A simple set-up of 3 Workers and a Coordinator on a LAN looks like the following figure where ‘C’ denotes the Coordinator and ‘W’ a Worker

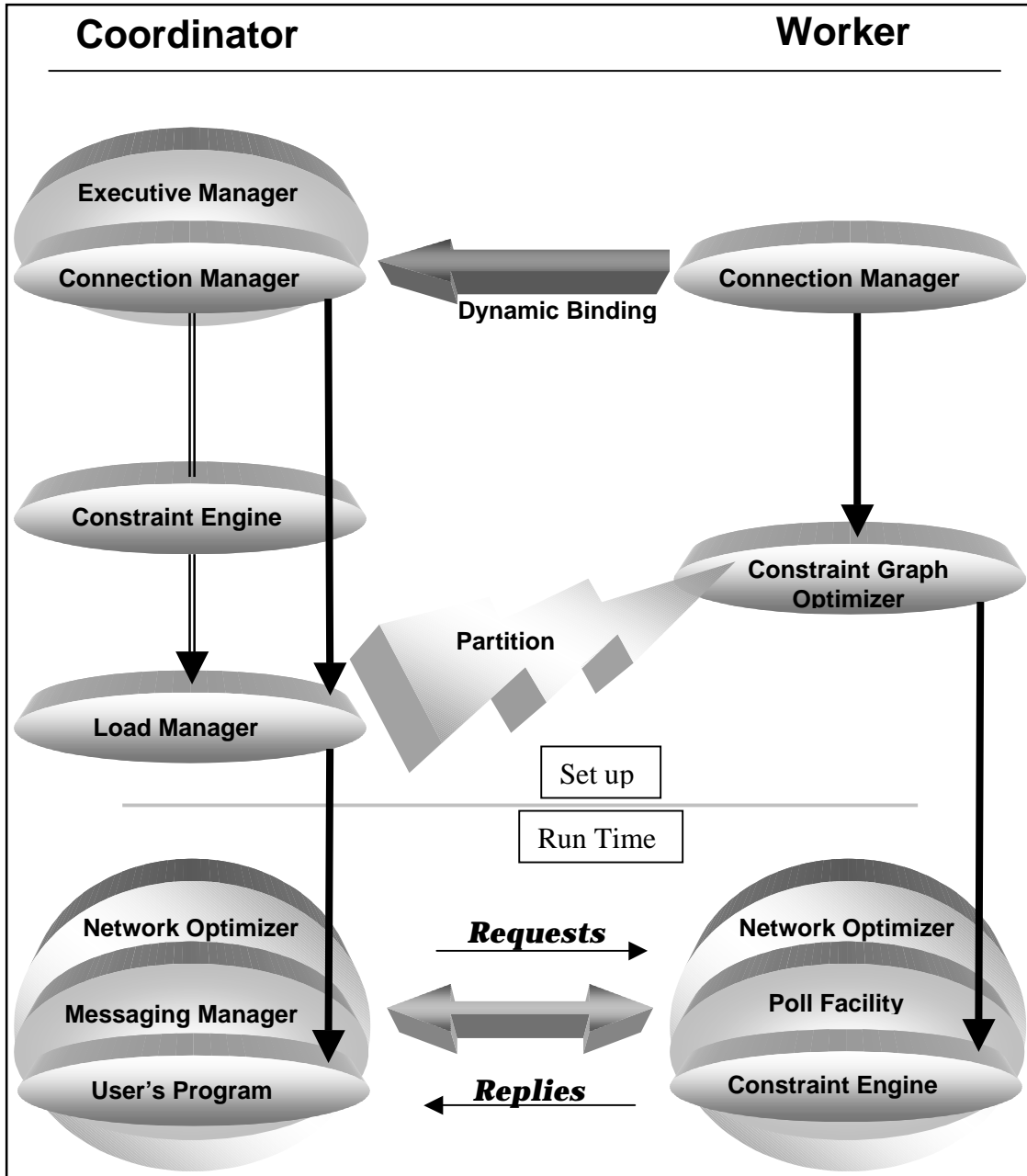


Coordinator and Workers on a LAN

High level system architecture

When a DLoVe program is executed in non-distributed mode, the `Link::InitCommunication()` and `Link::InitSystem()` calls do not do anything and simply return. But, when the program executes in distributed mode, these calls set up the entire execution environment for the program to run this mode. In the distributed mode, several modules (called “managers”) cooperate during this setup and at runtime both the Coordinator and each Worker initialize network communication (‘Set up’ section of the figure below). Runtime managers (‘Run Time’ section in the figure below) handle communication between Coordinator and Workers when the system is in execution. This communication always consists of requests from the Coordinator that invoke responses from Worker(s). There is no direct communication between Workers other than through the Coordinator.

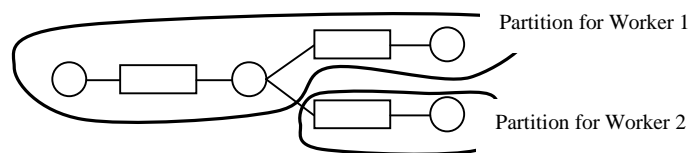
The ‘Executive Manager’ on the Coordinator is only involved when the system executes in Multi-user mode described in the following chapter.



DLoVe's managers

To execute a DLoVe program in distributed mode one must first invoke the Coordinator. Then the Coordinator accepts connections from Workers, as they are invoked on different workstations. This is managed in the “Connection Manager” that is part of both the Coordinator and Workers. The “Communication Manager” uses a configuration file to dynamically produce the socket binding between Coordinator and Workers. During this process, each authenticates itself, exchanging information with the Coordinator and establishes a communication channel.

Then the Coordinator partitions the graph (Link::InitSystem call) of Links and Variables, assigning a disjoint subset of Variables to each Worker using its “Load Manager”. These are the Variables a Worker will be “responsible for computing”. When each Worker receives its assignment, it executes its own algorithm in the “Constraint Graph Optimizer” manager to determine which sub-graph of the whole graph it will need to evaluate in order to compute its assigned Variables. For example, the following graph will be partitioned into two sub-graphs as shown below:



Partition into two sub-graphs

After partitioning is completed, the Coordinator starts executing the main application and the Workers start listening to requests from the Coordinator. The Coordinator reads all its input devices, requests Variables in parallel from the

Workers, and renders the screen. The Workers handle all requests from the Coordinator.

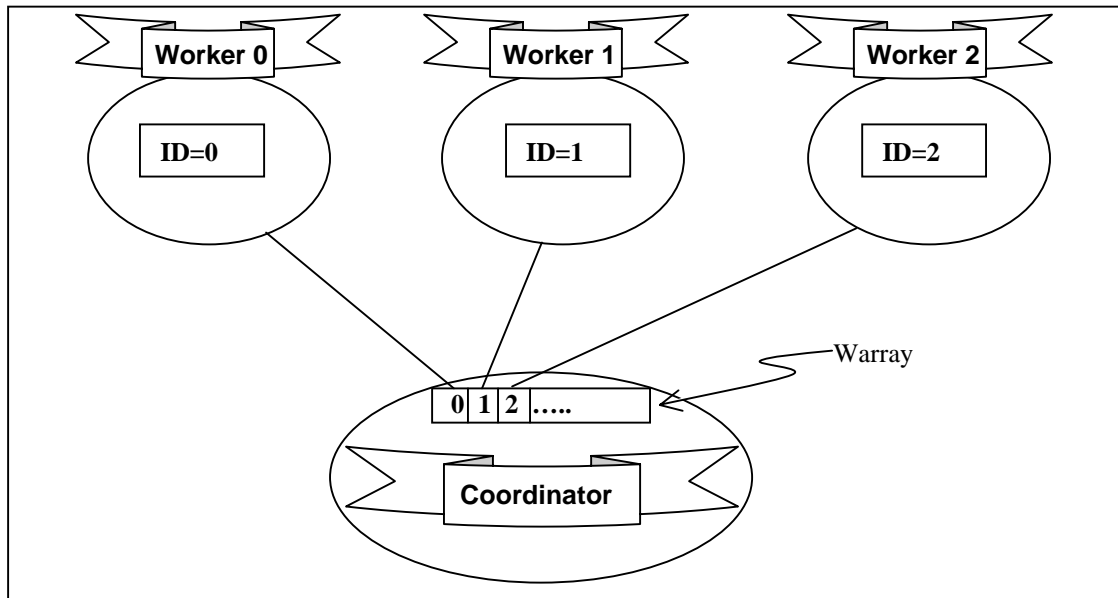
The last set of Managers, in the 'Run Time' section, handle all request-reply pairs between the Coordinator and Worker(s). The Coordinator executes the user's program. Using the "Message Manager" it incorporates requests into network messages that can be passed to Workers over the network. The "Network Optimizer" in the Coordinator is responsible for building large messages out of smaller ones, so that it can use the network more efficiently. It also makes sure not to overflow network capacity by sending too many messages over the network.

Critical messages, such as commands that enable or disable a Link, are always sent. Non-critical messages may be dropped if Workers seem overloaded. Each Worker uses the "Poll Facility" to get requests from the Coordinator and unpacks messages that may contain multiple requests. Then the "Constraint Engine" in the Worker processes all the GetE requests. In case there are several GetE requests that need to be returned to the Coordinator, the "Network Optimizer" builds a large message that contains several GetE replies, and sends it to the Coordinator as a reply in a single message.

The Coordinator also owns a "Constraint Engine" manager and thus it can perform any calculations it wants locally instead of requesting a Worker to do that for it. This is controlled by the programmer. We will see later in detail how the protocol works and how the messages look.

Communication Protocol

The very first call in the main() function should be Link::InitCommunication(). This call takes care of all the connections between the Coordinator and Workers. The configuration file supplies the DNS name or IP address of the Coordinator and the port number upon which the Coordinator is listening. Then the Workers connect to the Coordinator based on dynamic binding the same way ftpd works on a UNIX workstation. Every machine involved in running a program in the distributed mode must have a copy of the same configuration file. Each Worker requests a connection from the Coordinator. The Coordinator accepts and it then requests the Workers to reconnect back to it on different port numbers that it supplies and then disconnects. The Workers now know the new port number the Coordinator is listening on, and they reconnect (dynamic binding). There is a different port number for each connection. This process continues until all Workers connect to the Coordinator. The Coordinator knows how many Workers are going to connect because this information is supplied in the configuration file. When the system is running in non-distributed mode, this call does nothing, it simply returns. During this process, the Coordinator assigns a unique id to each Worker. This id is very important in the partition algorithm discussed later. The Coordinator holds an array (called Warray) of pointers to the connections to the Workers. The array index tells the Coordinator to which Worker this communication channel is connected as shown in the following figure:



Multiple Workers connected to a Coordinator

After the call to `Link::InitCommunication()` (line 2), the user creates the window(s) in the application (line 3) and all the Links and Variables (line 4). Then the `main()` function calls `Link::InitSystem()` (line 5). Any `GetE()` requests prior to the `Link::InitSystem()` call, which trigger the constraint solver, use the local constraint solver on the Coordinator to get the requested Variables up-to-date. The reason the Coordinator does not initiate requests to the Workers at this point is because the Coordinator has not executed the partition algorithm of the graph and thus, it does not know where to send the requests.

Workers, however, calls `Link::InitSystem()` and they never exit from this function. They stay in it in a loop, listening for requests from the Coordinator. The Coordinator, before exiting the `Link::InitSystem()` uses the “Load Manager” to partition the constraint graph and assign Variables to Workers. What is actually happening in the partition of the graph is that the Coordinator partitions the

queries. By doing that it knows which Workers are paying more attention to which Variables and it only asks the specified Workers for the specified Variables. The Coordinator knows where to request each Variable because it knows which Worker is responsible for which Variables. Even though all Workers, and the Coordinator, have exactly the same copy of the constraint graph in memory, and their constraint solver can bring any Variable up-to-date, Workers pay attention only to Variables assigned by the Coordinator. After the partition algorithm finishes, the Coordinator starts running the main application, 'Run Time' section, and any SetE(), GetE(), Link Enable/Disable requests, are formed into messages and sent as requests over the network to Workers.

Partition Algorithms

Besides the constraint algorithm there are two other main algorithms in DLoVe that take place in the "Load Manager". The first one is the partition algorithm performed by the Coordinator and consists of 2 phases. In this algorithm the Coordinator partitions the constraint graph and assigns the Variables to Workers. By doing this, the Coordinator actually partitions the queries so that it can request Variables in parallel. The Workers perform the second algorithm (optimization) in the "Constraint Graph Optimizer" after the Coordinator finishes the partition algorithm and transmits the partition information to Worker(s). The Worker(s) select the smallest set of Variables that they can pay most attention to so they can take advantage of their incremental constraint-solving algorithm.

Partition (2 phases) (Partition Queries)

The first phase of the partition algorithm marks each Variable with a number that indicates the number of dependencies. The second phase assigns variables to different workers in order, starting with the Variables with the most dependencies. The Partition algorithm is static, which means that it may assign easy tasks to powerful machines and difficult tasks to less powerful machines. It does not take into consideration the time needed to bring a Variable up-to-date or the speed of the machines on which Workers are running. A better approach would be to use a dynamic algorithm where the Coordinator would determine at run time the speed of each machine and partition accordingly. But because the constraint graph can change on the fly by enabling/disabling Links there is a big overhead for trying to re-partition the graph at run time. The pseudo-code of the Partition algorithm is shown below:

```

1  Partition( ) {
2
3      MaxVarsNeeded = -1;
4
5      foreach Variable var do
6          ClearALLVisited();
7
8          foreach Link link do
9              link->mark4partition = false;
10
11             Partition_phase_1(var, var);           // Modifies 'MaxVarsNeeded'
12
13
14     got_some = false;
15     int Wid = 0;
16     for ( i = MaxVarsNeeded; i >= 0; i-- ) { // i==0) for detached Variables
17         foreach Variable var do
18             if var->L_counter == i and not var->assigned then
19                 got_some = false;
20                 ClearALLVisited();
21
22                 Partition_phase_2(var, Wid); // Modifies 'got_some'
23
24                 if got_some == true then
25                     Wid = (Wid + 1) mod ParticipatingWorkers;
26
27     SendPartitionInfo_to_all_Workers();

```

The first part of the Partition algorithm (lines 3 - 13) loops around all Variables in the system and calls the recursive function `Partition_phase_10`. For each Variable before calling `Partition_phase_10`, the algorithm clears the `mark4partition` flag of each Link. This `mark4partition` flag ensures that the correct number of dependencies is counted for each Variable.

After the loop (lines 5 - 11) the algorithm knows how many Links are needed to bring each Variable up-to-date. The Variable `MaxVarsNeeded` (line 3), which gets modified by `Partition_phase_10`, is used to find out the maximum number of Links needed by Variables with the most dependencies. This variable is used later by the Partition algorithm in (line 16).

The second part of the algorithm (lines 14-25) is used to actually partition the constraint graph. The `got_some` Boolean variable indicates whether a Variable has been assigned to a Worker (true) or not, (false). The `Wid` Variable indicates the id of the Worker to which a Variable is going to be assigned. The 'for' loop (line 16 - 25) uses the `MaxVarsNeeded` variable. This loop first assigns the Variable with the most dependencies, the most Links that need to be evaluated in the worst case to bring a Variable up-to-date. Then the 'foreach' loop (lines 17 - 25) loops around each Variable. If the number of Links needed to be evaluated in the worst case is equal to the current number of the variable 'i' and the current Variable is not yet assigned to any Worker, the recursive `Partition_phase_20` function is called. This function assigns the current Variable, and recursively, all Variables in its dependency path to the Worker with id equal to `Wid`. If `Partition_phase_20` assigns any Variables to Worker with id equal to `Wid`, it sets the 'got_some' to true, and in line 25 selects the next Worker that will be assigned Variables, using a circular-like array. After all

Variables are assigned to Workers, `Partition()` communicates this information to the Workers so that they know to which Variables they have been assigned (line 27).

`Partition_phase_1()`, below, recursively finds the Variables that are not used as output by any Links. Then after it reaches the end of the tree in the graph (line 11), it unwraps the recursion. For each Link that has not been marked (`marked4partition` set to true), it increments the `L_counter` of the requested Variable, which keeps track of the number of Links needed to bring this Variable up-to-date. Lines 16 - 17 store the maximum number of Links needed by any Variable.

```

1  Partition_phase_1(output_var, on_this)
2
3      foreach Link link that uses output_var as output do
4
5          if not link->IsVisited() then
6              link->SetVisited();
7
8              foreach Variable var that is an input to Link link do
9
10                 Partition_phase_1(var, on_this);
11
12                 if not link->mark4partition then
13                     link->mark4partition = true;
14                     on_this->L_counter += 1;
15
16                 if on_this->L_counter > MaxVarsNeeded then
17                     MaxVarsNeeded = on_this->L_counter;

```

`Partition_phase_2()` uses a depth first search approach, as the previous algorithm, and assigns Variables to a Worker. It recursively traverses the graph to the Variables that are not used as output Variables by any Links, the leaves of the tree. The algorithm is shown below:

```

1  Partition_phase_2(output_var, Wid)
2
3      foreach Link link that uses Variable output_var as output do
4
5          if not link->IsVisited() then
6              link->SetVisited();
7
8              foreach Variable var that is input to link do
9                  if not var->assigned then

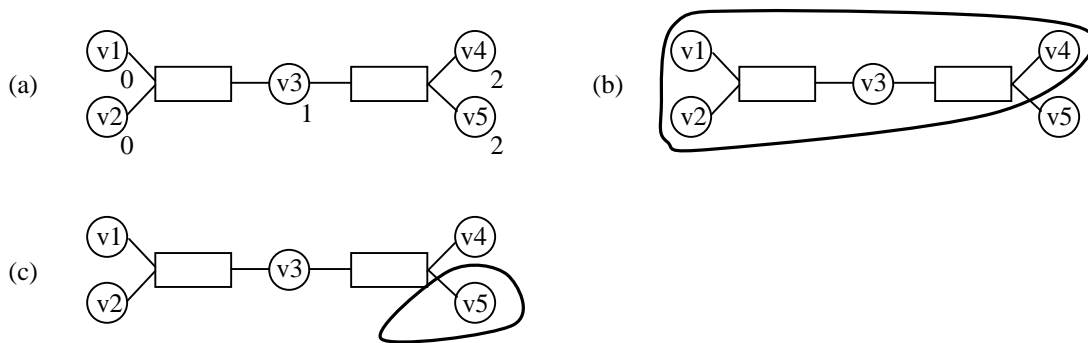
```

```

10         var->assigned = true;
11         CC->AssignVar( Wid, var->GetId() );
12         got_some = true;
13
14         var-> assigned_to_which_W = Wid;
15     fi
16     Partition_phase_2( var, Wid );
17
18
19     if not output_var->assigned then
20         output_var->assigned = true;
21         CC->AssignVar( Wid, output_var->GetId() );
22         got_some = true;
23         output_var->assigned_to_which_W = Wid;

```

The bottom section (lines 19 - 23) is the special case where only the original requested Variable is tested to be assigned to a Worker, in case that all other Variables upon which it depends, have already been assigned to a Worker. Note here that if any Variable of the dependency path of the original Variable is assigned to a Worker the `got_some` variable is set to true. Lines 19 - 23 are executed in graphs that look like (a) below (Links have multiple outputs):



The L_counter of the Variables

The numbers below the Variables in (a) show the `L_counter` variable of each Variable. This number is assigned to these Variables by the `Partition_phase_10` algorithm. With these Variables, the `Partition()` algorithm would call in descending order `Partition_phase_20` on either Variable 'v4' or 'v5', because they both have the

maximum `L_counter` among all Variables. Let us assume that 'v4' is selected and there are two Workers in the distributed environment. Then all Variables shown in (b) would be assigned to Worker 0. The next Variable that would be selected is 'v5' and the `Partition_phase_2()` would be called again this time on 'v5'. The 'if' statement (line 9) would prevent setting the `got_some` variable because all Variables that 'v5' depends on have been assigned to Worker 0 in the previous run. As a result the `Partition_phase_2()` would unwrap to the first call of itself that was called on Variable 'v5' in line 19. Since Variable 'v5' has not been assigned to any Worker yet, it will be assigned (line 21) to the next available Worker, Worker 1 as shown in (c).

For Worker 0 to bring 'v4' up-to-date it needs to evaluate both Links. So does Worker 1 to bring 'v5' up-to-date. Both Workers have the same semantic copy of the graph and they both know how to bring any Variable up-to-date. It does not matter that Worker 0 also got assigned Variables 'v1', 'v2', and 'v3'. We will see later that when Worker 0 runs its optimization algorithm, it will focus only on Variable 'v4'. Worker 0 knows that by bringing Variable 'v4' up-to-date 'v3' is also up-to-date. Worker 1 also brings Variable 'v3' up-to-date when it brings 'v5' up-to-date, and yes, there is some redundancy in DLoVe – and in this case total redundancy. However, when the Coordinator wants to request Variable 'v3' it will request it from Worker 0, since Worker 0 was assigned this Variable by the Coordinator.

Optimization (Workers) (MyMainVars)

The Workers in the “Constraint Graph Optimizer” Manager run the optimization algorithm. Even though the Coordinator may assign many Variables to Workers, Workers run this algorithm to find which Variables are the most important so they can pay more attention to them and take advantage of the incremental nature of the solver.

```

1  MainVarsSetUp()
2
3      MaxVarsNeeded = -1;
4
5      foreach Variable var in WW->MyVars do
6          var->L_counter=0;
7
8      foreach Variable v in WW->MyVars do
9
10         ClearALLvisited();
11
12         foreach Link l do
13             l->mark4partition = false;
14
15         Partition_phase_1(v, v);      // Pass v twice, Modifies MaxVarsNeeded
16
17     int i;
18     for ( i = MaxVarsNeeded; i > 0; i-- ) do
19         foreach Variable v in WW->MyVars do
20             if v->L_counter == i then
21                 MainVarsSetUp_phase_2(v);

```

The algorithm shown above first initializes the `L_counter` variable of every Variable similarly to what the Coordinator does in the Partition algorithm. This variable holds the number of Links that need to be evaluated in the worst case to bring the specified Variable up-to-date (lines 5 - 6). The foreach loop (lines 8 - 15) goes through all Variables that the Coordinator assigned to this Worker, clears the visited flag so it can detect and break cycles (line 10). Then it sets the `mark4partition` flag to false and calls recursively the `Partition_phase_1()`. Note that, this function is also used by the Coordinator in the Partition algorithm. The Worker calls this function because it also needs to know the maximum number of Links that need to

be evaluated for each Variable in the worst case; this information is stored in the `L_counter` variable in each Variable.

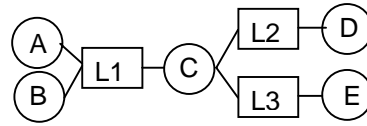
Then starting in descending order (line 18) it calls the `MainVarsSetUp_phase_20` function for every Variable that the Coordinator assigned to this Worker.

```
1 MainVarsSetUp_phase_2(output_var)
2
3   if output_var not in WW->MainVars then
4     foreach Variable var in WW->MainVars do
5       if output_var is a Variable that var depends on then
6         return
7
8     WW->MainVars->Add(output_var); // add it to the MainVars
```

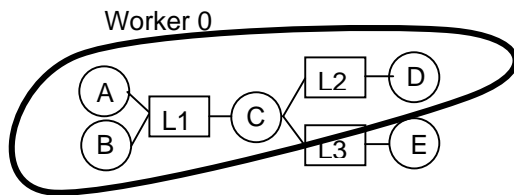
The `MainVarsSetUp_phase_20` collects all the Variables, from the set of Variables the Coordinator assigned to it, that are not needed by any Link in the graph as input. All these Variables are collected into the `MainVars` set (line 8).

Parallel Computation on a Distributed Graph

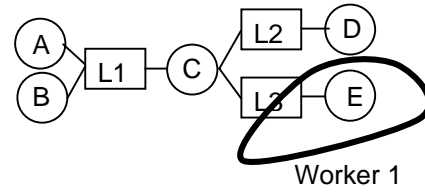
Every Worker and the Coordinator owns the same constraint graph representation. However, the Coordinator partitions the graph and assigns Variables to Workers. The Workers pay attention only to Variables that they have been assigned by the Coordinator. The set of Variables assigned to them represents a sub-graph of the entire constraint graph. For example, in the following example (a) where only two Workers are participating:



(a)



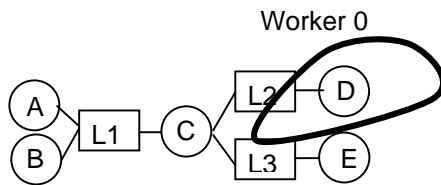
(b)



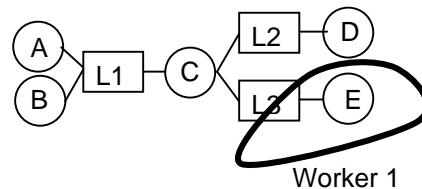
(c)

Partitioned graph to two Workers

Worker 0 will be assigned the Variables in (b) and Worker 1 will be assigned the Variable in (c). After the Workers run the optimization function, the Worker 0 will pay all its attention to Variable 'D' as shown in (a) below and Worker 1 to Variable 'E' as shown in (b) below. This way the Worker 0, for example, knows that by bringing the Variable 'D' up-to-date, all its Variables assigned by the Coordinator, are also up-to-date.



(a)



(b)

Partitioned graph after the optimization algorithm is run on the Workers

By partitioning the constraint graph, the Coordinator actually partitions the queries. By partitioning the queries, the Coordinator can initiate requests that are processed in parallel. For example, if the Coordinator calls $\text{varE} \rightarrow \text{GetE}()$ and $\text{varD} \rightarrow \text{GetE}()$, the requests will be processed by Worker 1 and Worker 0 respectively. Parallel processing on Variables is done specifically based on which Variable is demanded. Even though a $\text{varE} \rightarrow \text{GetE}()$ request could be processed by Worker 0, since both Workers have an identical copy of the constraint graph and both Workers run the same constraint solver, the Coordinator knows who is the best Worker to process this request. If the Coordinator initiates $\text{varC} \rightarrow \text{GetE}()$, it will be processed by Worker 0 because Worker 0 was assigned Variable 'C' and even though Worker 0 ran the optimization algorithm, it will process this request from the Coordinator. This gives room for fault tolerance where one of the Workers crashes, the Coordinator can detect this failure and re-direct requests to another live Worker.

Two flavors of constraint solvers

The Workers run their constraint solver when the Coordinator requests Variables. After the constraint solver finishes the Workers reply back to the Coordinator with the up-to-date values of the requested Variables. However, there are cases where the Coordinator needs to run its constraint solver and update Variables locally, instead of sending requests to Workers. This is appropriate when the partition algorithm is not yet run but the user needs some Variables to set up his/her program. The Coordinator does not know where to send these requests because the Variables have not yet been assigned to any Worker. In addition, the Workers are expecting Partition information at this point since this is how the DLoVe protocol

works. It also applies when the Coordinator wants to update critical Variables very fast. Such Variables in Virtual Reality include the position of the user's head and hand. Finally, it could also be used to support local variation as in Repo-3D [MacIntyre 98]

Constraint Solver on the Coordinator

To avoid motion sickness, and user frustration, any Virtual Reality system should support many frames per second. The head movement and the hand movement are an example of the most critical Variables that exist in a Virtual Environment. So, instead of sending requests to Workers and then waiting for replies and then updating the display, the Coordinator runs its local constraint solver to update such critical Variables. This is done as follows:

Instead of calling `varE→GetE()`, it calls the following two functions in order:

- `Do_evaluation(varE);`
- `varE→GetI();`

The `Do_evaluation(varE);` runs the constraint solver on Variable `varE` that brings it up-to-date, and then it gets its raw value by using the `GetI()` member function of the demanded Variable. It is alright to see a bouncing ball in the background a little bit jerky, but it is unacceptable for the user to turn his/her head or hand and have the display show the position of the head/hand with a two 2 second delay.

Constraint Solver on the Worker(s)

The Workers call the `Link::InitSystem()` and never exit from this function. They stay in an infinite loop listening for requests from the Coordinator. When a `GetE` request comes in, they run the `Do_evaluation()` function which is the constraint engine and they return the updated value of the demanded Variable to the Coordinator. The return value to the Coordinator is built into a reply message and is sent to the Coordinator. The user however, is still using the same `SetE()`, `GetE()`, `Enable()`, and `Disable()` functions. Their functionality is totally transparent to the user.

Configuration file

The configuration file is a text, colon ':' delimited file. The first field is the port number the Coordinator is listening for all Workers to connect. The second field specifies the DNS name or the IP address of the Coordinator, and the last field specifies the number of Workers that are going to participate in the program. A simple configuration file with 3 Workers is shown below:

```
PORT NUMBER:      2016
Master Coordinator:  mondrian.eecs.tufts.edu
Workers:          3
```

Chapter 8: Multi-user DLoVe

Introduction

Using the Links and Variables paradigm to program Virtual Environments gives the programmer the flexibility to execute in distributed mode a program that is designed to run on a single machine with minor code modification. Since the constraint graph is distributed and the Coordinator knows how to query/request Variables from the Workers and how to set Variables on the Workers, it is simple to add additional Coordinators to the picture to run multi-user programs. In fact, in a distributed system where there is a Coordinator and Worker(s), adding another user/Coordinator is simply specifying the number of additional Coordinators in the configuration file and running the additional Coordinators that are the same executable as the original Coordinator.

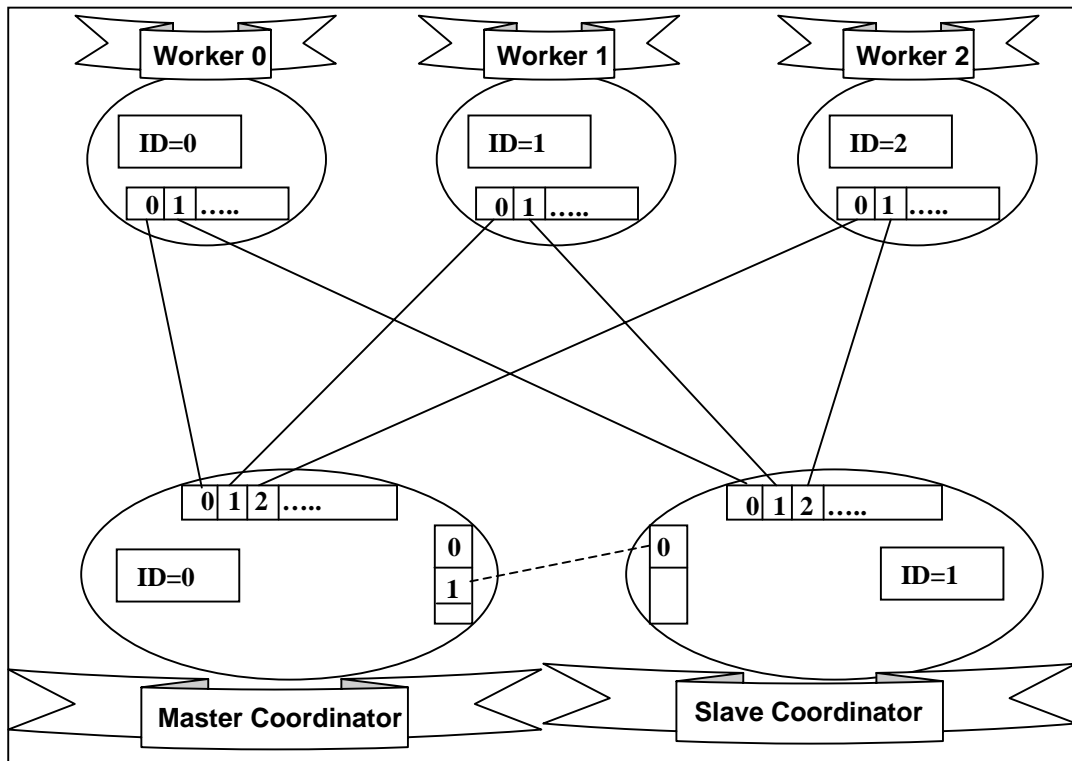
The DLoVe protocol does support multiple Coordinators in a DLoVe framework. However, there are several issues on how to run a multi-user program and I will

explain them shortly. The most important issue is that when more Coordinators are added to the framework, all Coordinators will have access to the same input Variables. That means that each Coordinator may control any other's hand, or any other input Variable that is attached to an input device. As we will see, one of the most important issues is to be able to specify that a specific device is attached to a specific workstation. The next issue is to create rules on how to transform a single user program into a multi-user program and assign roles to each user. There are some modifications that need to be performed to a program that is designed to run in a single user environment. And this is unavoidable since we need to give different responsibilities to each user and then each needs to know where the others are located in a virtual world.

Multiple Coordinators

In a multi-Coordinator environment, the first Coordinator is called the Master Coordinator, and all the other Coordinators are called Slaves. As with the Workers, all Slave Coordinators have to connect to the Master Coordinator and exchange information. The Master Coordinator always has id of 0 and the Slaves have an incremental id that is assigned to them by the Master, as with the Workers. The Slaves follow the dynamic binding paradigm, similarly to Workers, to connect to the Master. The Master keeps a separate array of pointers to channels to Slaves, and the index of the array indicates the id of the Slave. After all Workers and Slaves connect to the Master, the Master requests the Slaves to be servers and accept connections from the Workers on specified port numbers. It then requests the Workers to connect to each Slave following the same dynamic binding paradigm

when they connected to him the first time. After all Workers connect to the Slaves, all Workers inform the Master that they are connected to the Slaves and the main application starts. After this point the connections between all Slaves to Master are no longer needed and so they are terminated.



Multiple Workers connected to multiple Coordinators

The figure above shows the connections between all Workers to all Coordinators, and the temporary connection from the Slave to Master, which is terminated before the main application starts. The Coordinators have an array of pointers to communication channels to the Workers. The index in the array indicates the id of the Worker to which they are connected. They also have an array of pointers to communication channels between the Master and themselves. In the above figure,

the Master Coordinator uses the array index 1 which means that it is connected to Slave with id of 1. On the other end, the Slave uses the index 0 in the array to indicate that it is connected to the Coordinator with id 0, which is the Master. The Workers also have an array of pointers to communication channels to Coordinators. They are also using the index of the array to indicate which Coordinator they are connected to. The details on what kind of information is exchanged and what type of messages is passed back and forth are shown in the next chapter.

Issues – Transforming to Multi-user

The first issue to resolve is to identify devices that are attached to different machines. We need that so different users can interact in a virtual environment independently. The distinction between the different machines and thus the different input/output devices is done in the configuration file. In the configuration file, the IP address or the DNS name of a machine can be associated with an application level identification number. For example, the following lines in the configuration file

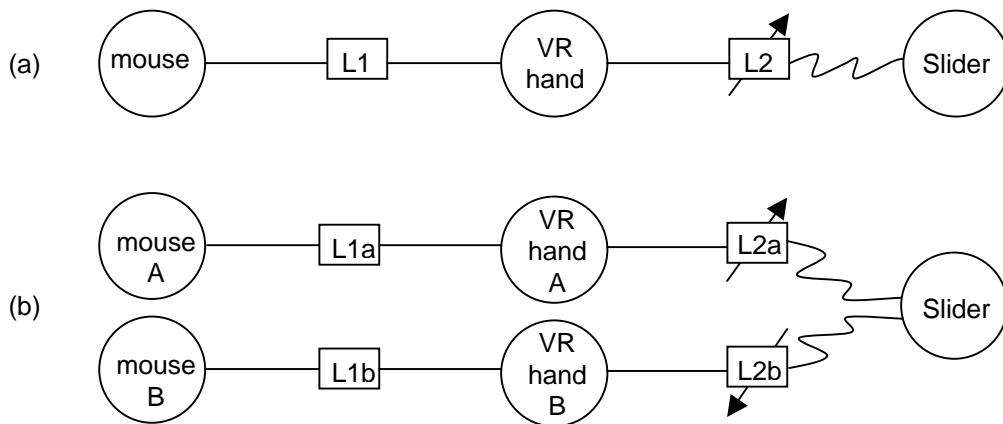
```
ID: mondrian.eecs.tufts.edu=4  
ID: vermeer.eecs.tufts.edu=3
```

associate the machine with DNS `mondrian.eecs.tufts.edu` with the application level id equal to 4 and the machine `vermeer` with id equal to 3. Depending on the number of Coordinators in the multi-user environment, the representing number of ID fields must be present. For example, one might specify that machine 'A' can grab only red objects and machine 'B' can only grab blue objects. Later, by swapping the

identification numbers in the configuration file and without any recompilation, machine 'A' can only grab blue objects and machine 'B' can only grab 'red' objects.

The second part of the problem is to actually be able to attach input Variables to same devices but on different machines. Let us say in a very simple program that the mouse controls a virtual hand on the screen that can grab and control a Slider. That can be represented in an aggregate form as shown in the following figure part

(a):



Distinguishing input devices

Using the token-based sub-system, the discrete model, we can test to see if the virtual hand intersects with a slider and if the user presses the left mouse button then Link 'L2' becomes enabled and the slider follows the virtual hand (a). In a multi-user environment, each user needs to see where the other user is, or at least where the other user's hand is. So we need to create two sets of virtual hands, one for user 'A' on machine 'A' and one for user 'B' on machine 'B'. We can specify that the mouse attached on machine 'A' is called mouse_A and the mouse attached to

machine 'B' is called mouse_B, using the ID: field in the configuration file. At this point the user on machine 'A' can do VRhand_A->GetE() to find its position and draw itself. It can also do VRhand_B->GetE() to find the other user's position so it can draw that as well. Similarly the user on machine 'B' can draw his hand and also the other user's hand.

The first rule is that every Link that uses the VRhand Variable as input, needs to be duplicated along with all its dependencies. The output of such Links needs to be the same. For example, in (b) 'L2' needs to be duplicated into 'L2a' and 'L2b' that use the Variable Slider as output. However, 'L2a' uses as input Variable VRhandA and 'L2b' uses as input Variable VRhandB. Depending on which system grabs the Slider, the representing L2 Link gets enabled. If both users grab the slider, then both L2 Links are enabled and both users can move the Slider. If one user moves the Slider to the top and the other to the bottom, DLoVe is going to execute both constraints and the users will see the Slider jumping up and down. This can be fixed by using a switch mechanism where the programmer can specify that only one user is allowed to grab the slider. An example of a switch is shown in another chapter. When a switch is involved, then if user 'A' grabs the slider from user 'B', the Slider is taken away from user 'B' and user 'B' does not control the position of the Slider any longer. User 'B' has to re-grab the slider to gain control over it. But this mechanism is not present in the single user environment, and additional code needs to be written.

All the events sent by the discrete subsystem are relabeled to indicate which machine caused the event. For example, in non-distributed mode when the right mouse button is pressed the event RMbuttonPRESSED may be sent. In the distributed environment, the discrete subsystem sends the event

RMbuttonPRESSED_3 if the machine with application level id equal to 3 caused this event or RMbuttonPRESSED_4 if the machine with application level id equal to 4 caused this even. This way the Coordinators know which Link to Enable/Disable and send the correct message to the Workers. The programmer does not see how the events get re-labeled. The programmer only needs to know that events get re-labeled so he/she can write the proper code to catch the events and take the correct action. There are cases however, where we do not care which machine initiated the event. For this reason the discrete subsystem also sends the original event before it gets re-labeled.

Backward Notification of Events

There are cases where we want to specify things like user 'A' can only grab red objects and user 'B' can only grab blue objects. But with specific action or gesture the behavior should change and user 'A' can only grab blue objects and user 'B' only grab red objects. And with another event, both users can grab both kinds of objects, blue and red. Let us say for simplicity that when the character 't' is pressed whoever could grab blue objects can now only grab red objects and vice versa. And when the character 'T' is pressed both users can grab both kinds of objects. Note here that both users can press the 't' and 'T' character. If user 'A' is moving a blue object and user 'B' types the 't' character, user 'A' must immediately not be able to move the object he was moving because he is now only allowed to grab and move red objects.

We need a mechanism to immediately notify the other Coordinator of this event. Because the Coordinators at run time do not communicate directly with each other but rather via the Workers, we could set a Variable on the Workers so that when the other Coordinator queries the Worker it will see that that Variable has changed and take the appropriate action. But how often should we query the Workers for this specific Variable. The 't' event is a discrete event and does not happen very often. That was the main reason I separated the UIDL into continuous and discrete subsystems.

To implement this behavior a new SetE() function needs to be used called SetE_Broadcast() and it is used for backward notification. When a Coordinator needs to set a Variable that would indicate a specific event, in this case that the character 't' was pressed, it uses this new one function SetE_Broadcast(). This function works the same way as the SetE() function but it also instructs the Worker to return in a reply message the result of the Variable that just got set, to all other Coordinators in the system. Even though all Workers get the SetE requests, only one of them will send this backward notification to all other Coordinators; only the Worker that is responsible for this Variable. The Coordinators will receive this notification as if they requested this Variable with GetE; they will not notice the difference.

The Need for Local Evaluation

In the single user distributed DLoVe framework, the position of the head and the hand were the most critical Variables and thus they were updated locally on the

Coordinator. In a multi-user environment however, each user needs to see where others are to work in a collaborative environment. Each Coordinator must update the head and hand position locally and also send that information to the Workers so the Workers know where each Coordinator is. Then each Coordinator can request the position of the head and hand of all other Coordinators with the GetE() member function. This way the local user does not feel any delay on the camera, which is moved by the head and it also informs the Workers on its location so they can reply back to the other Coordinators when they are queried. Some delay or jerky behavior of the position of the other users' head and hand is acceptable, since it is not that critical. Another use of local Evaluation is a limited form of Local Variation. When the user intersects his/her hand with an object that he/she can grab, since there is no feedback to tell the user if the object has been touched by his/her hand, we need to highlight the object to indicate intersection with the virtual hand. The other users do not have to see that the object is highlighted however. Highlighting an object is an operation local to the user to assist him/her see that the object in consideration has been touched.

Time is a special Variable

When the system runs in multi-user mode, time sensitive simulation becomes a problem. Since the Coordinator sets the time, in multi-user mode all Coordinators try to set the timer according to their own clocks. This may come in conflict with the current setting of the timer. To avoid this conflict only the Master Coordinator sets the clock and all other Coordinators get the time value through the Workers. So computer simulation objects are consistent among all Coordinators since they

share the same clock. Time sensitive object behavior is for example, the tossing of a ball, where its position is calculated based on time, speed, and acceleration. If two machines had different clocks, then one machine would set the clock to 30 the other to 100 and the position of the ball would move back and fourth depending on whose timer it was using. By having a single shared clock, all simulated objects are consistent among all Coordinators.

Chapter 9: Messaging (DLoVe Protocol)

Introduction

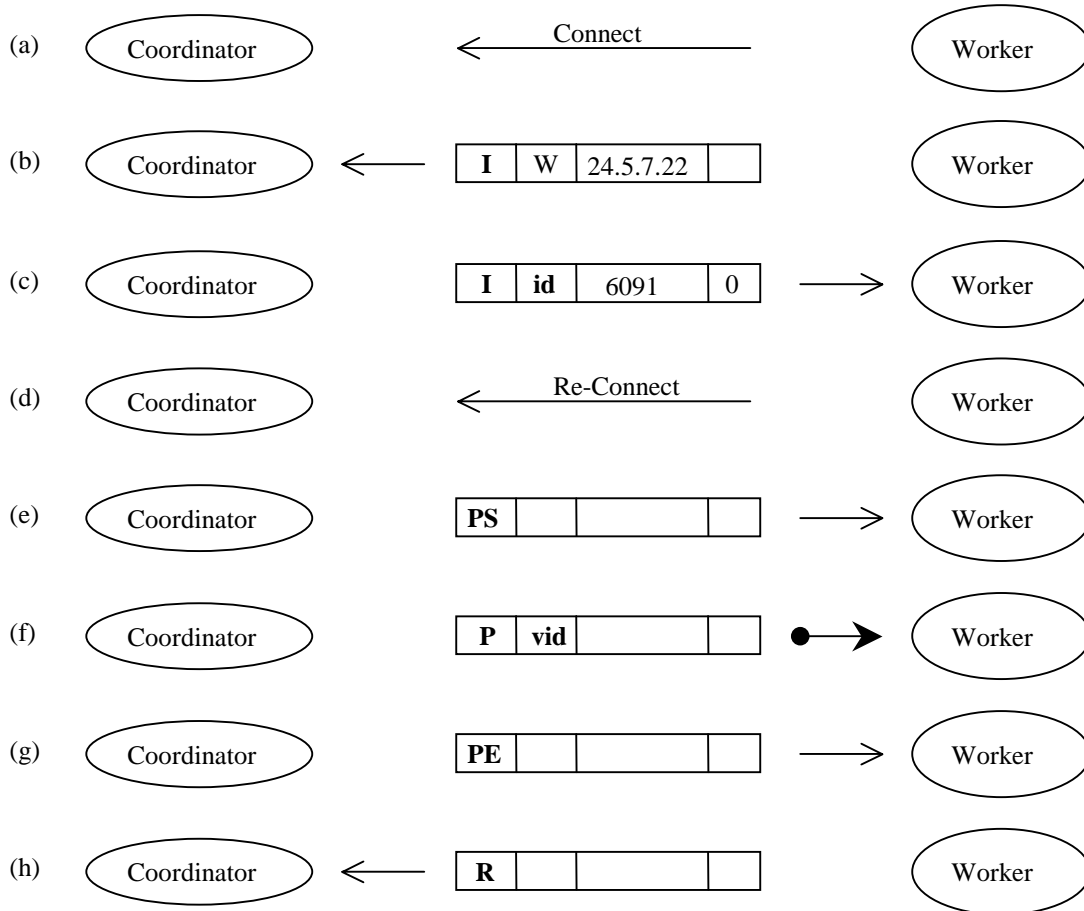
The first thing I will explain in this chapter is how all the machines get connected before main program starts, and then the type of DLoVe messages that are going over the network. DLoVe uses a message passing mechanism to communicate to all systems in its environment. There are three flavors on the format of messages sent to Workers from the Coordinators: Single messaging, Multi-messaging and blocking for replies, and Multi-messaging without blocking for replies, also called asynchronous mode. Even though, only the third one is recommended, I will describe all of them and explain how I arrived at them and why the first two do not work. Finally I will show how the same protocol works for multi-user environments with some code modifications.

Machine connections in Distributed Mode

The first machine/process that must come up before any other is the Coordinator. The Coordinator starts listening on a port number provided in the configuration file. The configuration file also provides the number of Workers that will connect to Coordinator. Then all the Workers connect to the Coordinator using dynamic binding. This way the entire set of machines can connect to a single port and the rest is left up to the Coordinator to orchestrate. The messages between the Coordinator and the Workers have four fields. The first field indicates the type of a message. 'I' means information, 'PS' means that the Partition information is to follow, 'P' indicates Partition information, 'PE' means that the Partition has finished, and 'R' means that the system is ready to accept and process requests. The rest of the fields may be null or if they are used they are explained below. The arrows indicate the direction of the message. Both the Coordinator and the Workers block after each message they send in a synchronized way.

In detail, first the Coordinator comes up and starts listening for connections and then the Workers connect one after the other (a). After a Worker connects, it sends a message to the Coordinator that indicates that it is a Worker (second field) and its IP address (b). Both fields are not needed in the distributed mode but they are needed in the multi-user mode, and for the protocol to be consistent Workers supply this information to the Coordinator. Then the Coordinator sends a message to the Worker that includes newly assigned id, a new port number for the Worker to reconnect and also the Coordinator's id; which is always 0 (c). Here is where the Coordinator assigns ids to Workers. The fourth field is always 0 since in the distributed mode there is only one Coordinator. Later in the multi-user mode, other

Coordinators place there their own id. The connection Worker – Coordinator gets terminated and the Worker requests another connection to the Coordinator on the new port number it received (d). This is the end of the dynamic binding. The Coordinator goes through this process for every Worker assigning new ids and providing new port numbers for them to connect. After all Workers connect to the Coordinator, the Coordinator sends a message to all Workers informing them that the Partition process is going to start (e). It then runs the Partition() algorithm and for every Variable it sends a message to the Worker that gets assigned this Variable a message (f). All Variables have a unique id among all systems that participate in the distributed environment. This id gets assigned to each Variable based on the order of creation. Since all Variables get created before the Partition algorithm is run, Workers and Coordinators have the same Variables with the same ids assigned to them. After the Partition() algorithm finishes, the Coordinator sends another message to all Workers indicating that the Partition reached its end (g). The Workers at this point run their optimization algorithm and when they are ready they notify the Coordinator (h). The Coordinator in the mean time is waiting for this message from every Worker. When every Worker is ready the Coordinator starts running the main application.



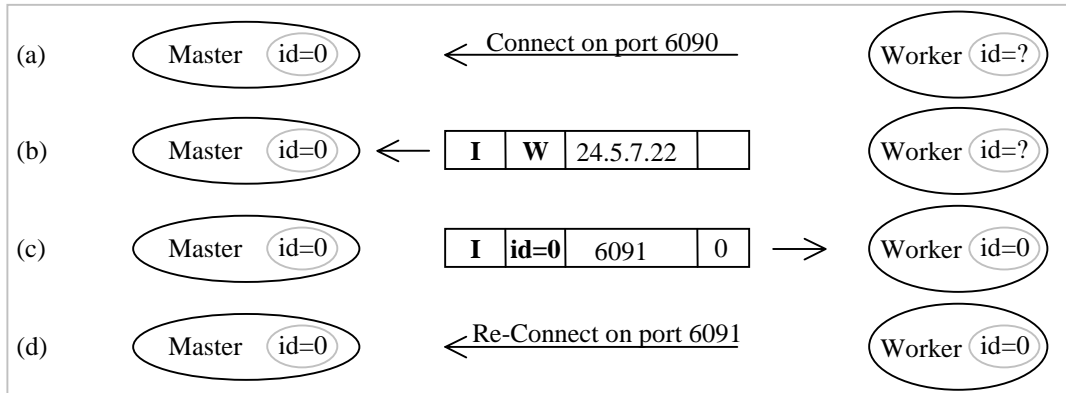
DLoVe's Connection Protocol

Machine Connections (Multi-User)

In a multi-user environment there are more than multiple Coordinators and thus the protocol is a bit different. The first Coordinator that starts is called the Master Coordinator and all the others Slave Coordinator. In the configuration file the IP address or the DNS name of the Master Coordinator is specified, and the number of

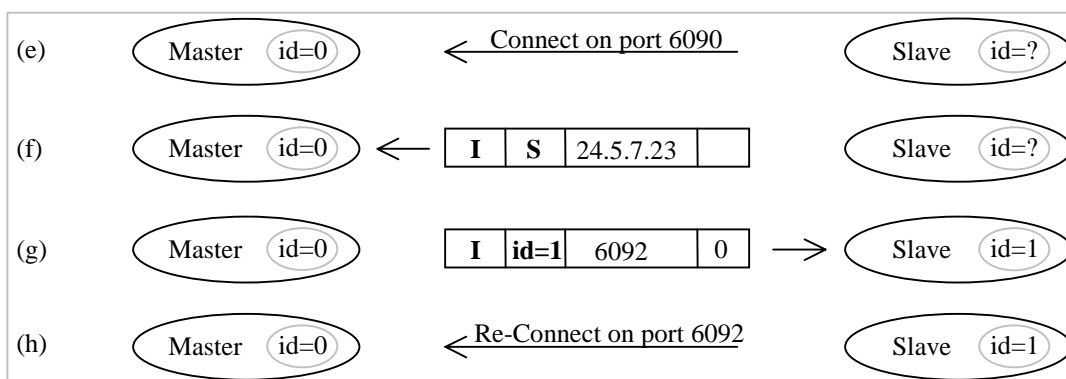
Slaves that are participating in the multi-user environment. Later the Workers need to connect to all Slaves but the Master coordinates all this as it is shown below. The Master and the Slaves run the same executable and their behavior is almost identical. They are mostly different in the initialization process where all systems connect with each other. The following example, for simplicity purposes, involves two Coordinators and one Worker. The port number that the Coordinator is listening on is 6090, which is also specified in the configuration file. The id number assigned to each machine is indicated in a little circle next to the machine type.

First the Master Coordinator starts up and then the Worker connects to it on port 6090 as shown in (a). The Worker then sends a message indicating that it is a Worker process requesting the connection, the 'W' in the second field (b). In the third field it specifies its IP address. Then the Coordinator sends a message that assigns an id to the Worker, which is zero in the second field (c). The new port number that the Worker should re-connect, port 6091, is specified in the third field. The fourth field indicates the id of the Coordinator itself, which is zero since it is the Master Coordinator. In (d) the Worker re-connects on the new port number, which is 6091. The Coordinator gets the initial port number from the configuration file; it then requests clients to reconnect to a new port number that is an increment of the initial port number.



Worker connects to Master Coordinator using dynamic binding

Then, using the same procedure the Slave Coordinator connects to the Master Coordinator (e). The Slave sends a message that specifies that it is a Slave Coordinator requesting a connection using 'S' in the second field, and in the third field supplies its IP address (f). In (g) the Master assigns an id to the Slave which is one, in the second field, since it is the next Coordinator in the system - zero is always the Master. The third field contains the new port number to which the Slave should reconnect, and the fourth field is the Master's id. In (h) the Slave reconnects on the new port number which is 6092.



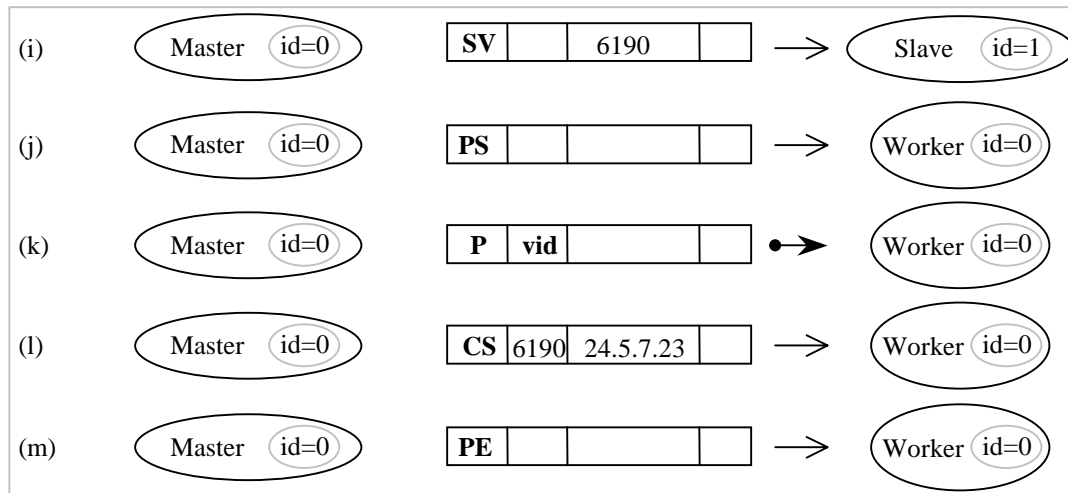
Slave Coordinator connects to Master Coordinator

After the Slave connects, the Master sends a message requesting the Slave to start listening on port 'initial port number from configuration' + 100 (i). In a while the Master will request the Worker to connect to the Slave. So the Slave should be listening for incoming connections. The reason the port number is an offset of 100 from the initial port number to allow multiple coordinators to run on a single machine in case the programmer wants to run some tests while developing a software.

Then the Master starts the partition process (j). Then it partitions the graph and sends the information over to the Worker (k). The vid in the second field is the id of a Variable that this Worker was assigned. In this example all Variables will get assigned to this one Worker, but in practice when multiple Workers participate each one will get a sub-graph of the original graph.

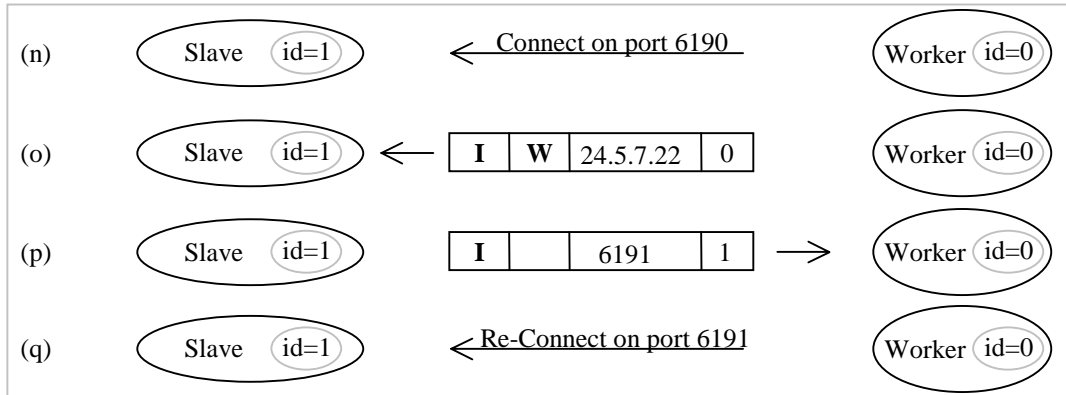
In (l) the Master requests the Worker to connect to the Slave Coordinator. It provides the port number the Slave is listening on and its IP address.

After that the Master is done and sends an 'End of Partition' message in (m). After this message is sent out, the Master waits until all the other systems are ready to proceed.



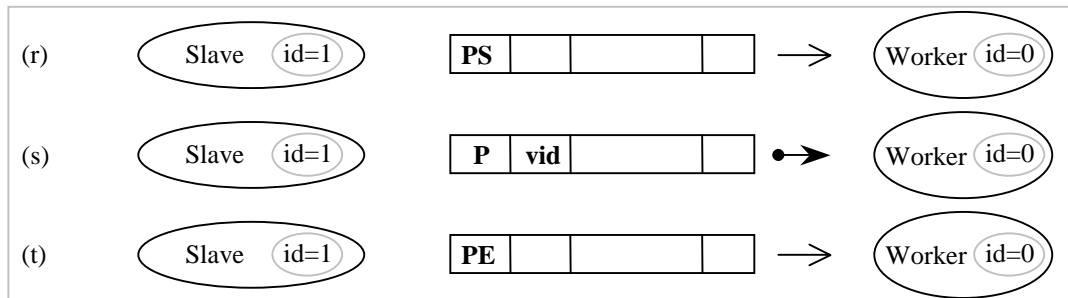
Slaves start listening for connections, Master partitions the graph, and the Workers are requested to connect to Slaves

Then the Worker connects to the Slave Coordinator (n) on port 6190. Then the Worker sends its IP address, and also in the fourth field its id, which is 0 (o). The Worker has to tell to the Slave its id because the Slave is not allowed to assign ids to Workers. The Slave in (p) now knows that a Worker with id of 0 is connecting and does not need to assign a new id and thus the second field is null. The third field is the new port number the Worker should reconnect and the fourth field is the id of the Slave which is 1.



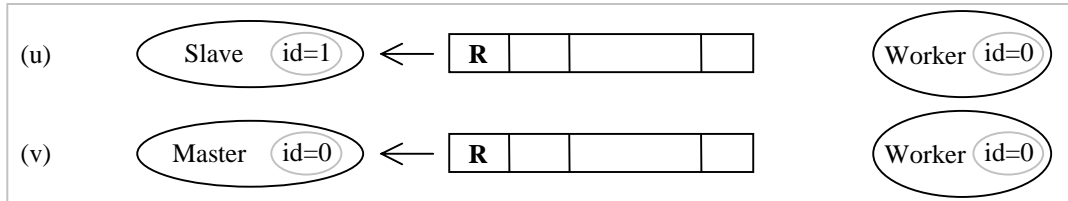
Worker connects to Slave Coordinator using dynamic binding

In a similar manner to the Master, the Slave Coordinator starts the Partition (r). In (s) it partitions the graph and sends the information over to the Worker. The Slave is not allowed to modify the partition of the graph. The graph was partitioned once by the Master and that is how the partition should be. However, the Slave partitions the graph and sends the information over to Worker for double-checking. If there is any discrepancy, an error is printed and all machines terminate. It is not possible to have Coordinators that are working on different partitioned graphs. In (t) the Slave sends the 'End of Partition' message to indicate that it is also done, and it is waiting for any other system to get ready so they can all proceed.



Slave Coordinator partitions the graph (just for verification)

At this point both Coordinators are waiting a message from the Worker to tell them that it is ready to process requests. The Worker runs its optimization algorithm and then it notifies both Workers that it is ready to process requests (u) and (v).



Worker is ready to process requests and so it notifies the Coordinators

Now the multi-user program starts and the Coordinators request Variables from the Worker. The Worker on the other is listening and satisfies requests.

Flavors of Message-passing in DLoVe

There are three configurations for message passing. The first one is the Single Messaging, where each request is a single TCP packet. Each packet is sent to the Workers in a non-blocking fashion with the exception of GetE requests. When the Coordinator requests a value of a Variable, it blocks until the Worker replies back to the Coordinator with the value of this Variable. The second configuration is similar to the previous one. GetE requests work the same way. However, all other requests that include SetE, Enable, and Disable, are packaged up in a big message and sent to Workers. In the third configuration, is the most appropriate for TCP/IP. The

Coordinator packages all requests, including GetE, in a big message and sends them over to Workers. The Coordinator does not block for GetE replies. It keeps sending requests and gets them back when they arrive. Until the requests comes back, the Coordinator uses the previous values of the Variables. Because there is a danger that the Coordinator may overflow the network, it keeps count of how many pending messages there are on the network. If the pending requests cross the threshold, the Coordinator drops the requests. The only exception is when the messages contain Enable or Disable requests, because these re-wire the graph.

Single messaging

In Single messaging each Message has four fields:

- type
- size
- id
- value

The 'type' field indicates the type of request, which includes SetE, Enable, Disable, or GetE. The 'size' field indicates the number of bytes in the 'value' field. The 'id' field is to indicate on which Variable (for SetE and GetE requests) or Link (for Enable and Disable requests) this request is to be performed. The last field, 'value', holds the value of a Variable. When a SetE request is sent the 'size' field indicates the number of bytes in the 'value' field. When Enable/Disable/GetE requests are initiated, the 'size' field is zero because there is no value in the 'value' field. For GetE requests, the Worker returns the updated value of the demanded Variable in a same format message. When the Workers reply, they put the value of the demanded

Variable in the 'value' field, the number of bytes of the value in the 'size' field, and send the message to the Coordinator.

For every request, the Coordinator builds a message, puts the request in the message and sends it over to the Workers. For SetE/Enable/Disable requests the Coordinator does not expect any reply from the Workers. However, for GetE requests it does. When the Coordinator sends a GetE request, it blocks until the Worker replies back with the value of that Variable. Sending many small packets using TCP over the Ethernet is not very efficient for the network [Halabi 97] [Lewis 98] [Lewis 98]. For simple programs it seems to work, but for Virtual Reality programs where many Variables are changing very fast, this configuration is unacceptable. Performance is so low that some times the user thinks that the program is not even running because the Coordinator requests many Variables and blocks for each one of them.

type	size	id	value
SETE	8	46	10011....

(a)

type	size	id	value
GETE	0	47	NULL

(b)

type	size	id	value
ENABLE	0	33	NULL

(c)

type	size	id	value
DISABLE	0	33	NULL

(d)

type	size	id	value
GETE	16	47	101001....

(e)

Requests made into Messages

In the figure above, (a), (b), (c), and (d) are the four types of messages sent to Workers from the Coordinator. (e) is the reply message sent by the Workers to a GetE request.

Multi-messaging - Block for replies

In this configuration, the Coordinator builds a multi-message that consists of several SetE, Enable, and Disable requests and sends this multi-message to Workers. However, as in Single-messaging, GetE requests are sent following the Single-messaging mechanism. The Coordinator still blocks to get the reply from the Worker to which the request was sent. The multi-message is patched with a header so that the Workers know how many requests are included, and how large the multi-message is. The multi-message header includes the following fields:

- nRequests
- mSize
- buffer

The 'nRequests' indicates the number of requests in the multi-message. The 'mSize' indicates the size of the 'buffer' field. All the requests are stored in the 'buffer' field. The 'buffer' field consists of multiple Single messages the same as those in Single-messaging.

nRequests	mSize	buffer							
5	102	type	size	id	value	type	size	id	value
		SETE	8	46	10011....	DISABLE	0	33	NULL
		type	size	id	value	type	size	id	value
		SETE	8	46	10011....	SETE	8	46	10011....

Multi-message containing multiple mini-messages

Even though this mechanism is increasingly faster than Single-messaging, it is still slow and unacceptable for Virtual Reality. Its slowness is because the Coordinator still blocks while waiting for replies from the Workers. It is faster than Single-messaging because it better utilizes the network by sending fewer but bigger messages over the network. All the requests in the 'buffer' field are the same as the ones in Single-messaging.

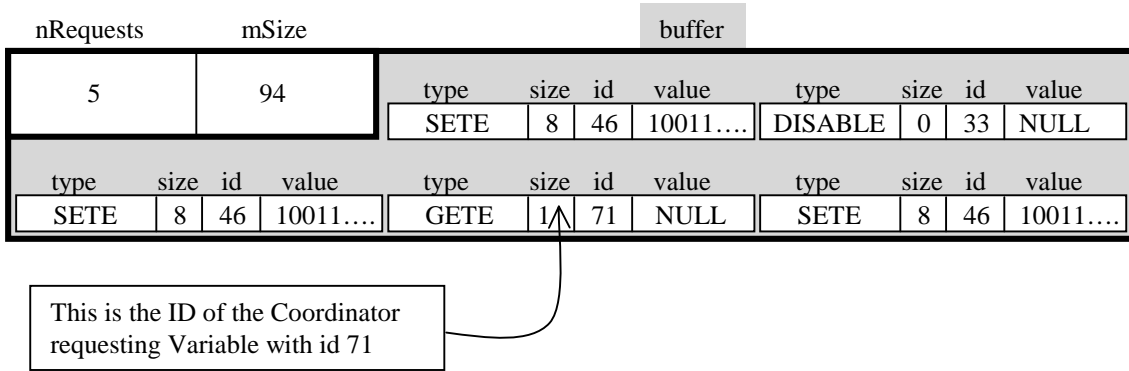
Multi-messaging - Non-Block

The next mechanism of sending messages is similar to the previous one. Here the Coordinator builds again a multi-message and this time even the GetE's are included. The Coordinator does not block for replies however. It rather sends the requests, and when the GetE replies come back it uses them. In the mean time, it uses old values of the Variables. In Virtual Reality this is acceptable as long as the delay is not too big. Using this asynchronous approach introduces several problems. The first one is that the Coordinator keeps sending requests without blocking for replies and this may overflow the network and as a result the application crashes. The Coordinator needs to know when the network is getting

near the overflow threshold so that it stops sending messages. That is why there is a tunable parameter in the configuration. This parameter indicates the threshold line of how many pending messages/requests are allowed on the network. The Coordinator keeps building the messages but if it already sent too many messages, crossing the threshold, it then disregards the messages instead of sending them to the Workers. This, however, causes another problem. What happens if the message that was disregarded included an Enable/Disable request? Enable/Disable requests modify the constraint graph on the fly. If one of these messages does not get to the Workers, the Workers will work on the wrong constraint graph. That is why before the Coordinator disregards a message it checks its mini-messages, in the 'buffer' field, and if it includes Enable/Disable requests, it sends them to the Workers even if it crosses the threshold. Enable/Disable requests are of high priority and cannot be disregarded.

There is also another problem with the GetE requests. In the previous two messaging mechanisms when the Coordinator sent a GetE request, the Worker got it, ran the constraint engine and replied back to the same Coordinator on the same communication channel where it received the request. In the non-blocking mechanism however, all Workers see the GetE requests. They are smart enough to check to see if they are responsible for each GetE request however. If they are not, then they disregard the requests, otherwise they reply back to the Coordinator. But what happens when there are several Coordinators in case of a Multi-user environment. The Workers need to know which Coordinator initiated the request so they can reply back to that Coordinator. To solve this problem, when the Coordinator initiates a GetE request, it places its own 'id' in the unused 'size' field of the mini-message. This way, Workers know who initiated the request and reply back to the requestor. In an environment where we have two Coordinators, one with

id=0 and the other with id=1 and the Coordinator with id=1 requests the Variable with id 71, a sample multi-message would look like the following figure:



GETE requests embedded into the multi-message

Configuration file

The configuration file is a text file that contains several colon-delimited fields. This file is shared among all Coordinator(s) and Worker(s). It contains the following:

- The number of Workers in the system
- The number of Slave Coordinators when in Multi-user
- The IP address or DNS name of the Master Coordinator that eventually all systems will connect to.
- The Port number of the Master Coordinator that is listening on
- The IDs of each Coordinator when in Multi-user mode (to individualize Coordinators)
- The flavor of the message-passing algorithm.
- The threshold of how many pending messages can be on the network

The only information the Workers need is the IP address or the DNS name of the Master coordinator and the Port number upon which the Master is listening. The

rest is communicated to them via the Coordinator by the DLoVe protocol. For simplicity however, all machines participating in the DLoVe environment are provided with the same copy of the configuration file. A complete description of the configuration file is provided in Appendix B.

Chapter 10: The ‘arms’ Application

Overview

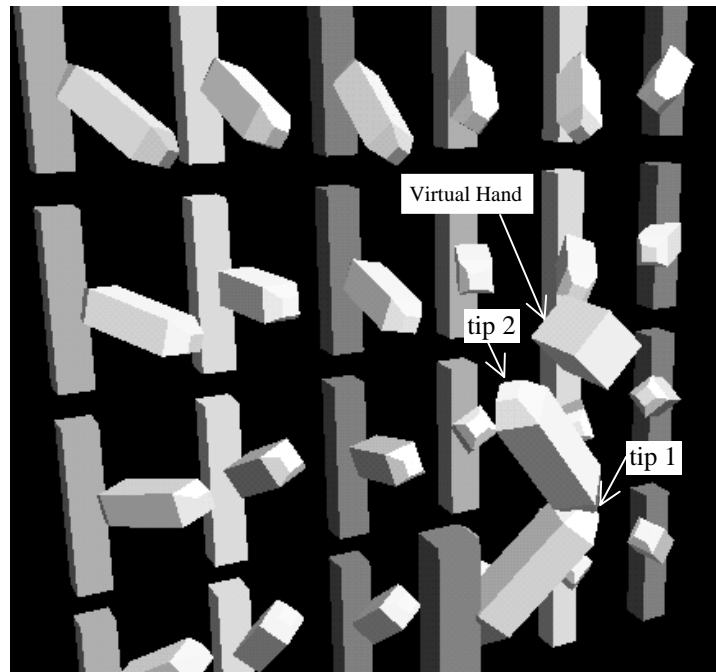
DLoVe was designed to address a variety of issues inherent in the specification and implementation of non-WIMP user interfaces. Its level of success can be measured directly or indirectly by attempting to create applications that rely upon the purposed features of the UIMS. The lessons learned from the development exercises, combined with a concrete measure of how easy and correct the application performed, may be interpreted as an indicator of the success or failure of the UIMS itself.

Over twenty complete applications and a variety of code fragments have been developed to test the correctness and robustness of DLoVe. From this set of applications, several sample development efforts will be presented in detail in this

chapter. For every application I present, I will explain how it can also be transformed to run in a multi-user environment.

The 'arms' program consists of two sets of arms. The first set of arms consists of one double-jointed arm, an arm that has attached to it to movable sub-arms, called Arm2. The second set of arms consists of twenty-four single-jointed arms, called ArmSlaves. The ArmSlave arms follow the direction of where the Arm2 is pointing. The user can grab any of the two sub-arms of the Arm2 and move them. Half of the ArmSlave arms follow the first tip of the Arm2 and the other half the second tip.

Initially the 'arms' program was designed to run on a single machine for a single user. With no modifications, the same program can run in a distributed mode evaluating the Links in parallel. Lastly I will show how the arms program was modified to run in a multi-user environment and how roles can be assigned to each user participating in the virtual environment. A screen dump of the program in execution is shown in the following figure:



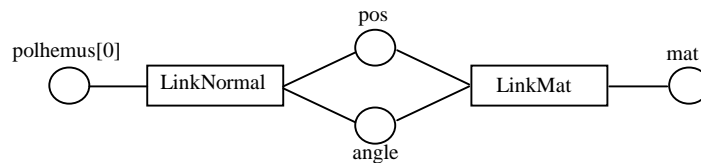
Snapshot of the 'arms' program

The Arm2 arm is shown in the foreground of the figure. And all the ArmSlave arms are shown in the background. Half of the ArmSlave arms are pointing to the first tip of the Arm2 arm marked as “tip 1” in the screen dump above and the other half at the second tip of the Arm2 arm, marked as “tip 2”.

Objects in 'arms' - single user.

In the 'arms' program the user can grab any of the two sub-arms of the double-jointed arm and move them. The hand of the user is shown in the scene graph as a small cube. The Polhemus sensor is sending six floating point numbers that indicate the position and angle of the hand/sensor in space. In every cycle in the

main program the polhemus[0] Variable gets set with the SetE operation with what the Polhemus sensor sends. Then the Link 'LinkNormal' extracts the 3D coordinates and the 3 angles and places the result in Variable 'pos' and 'angle' respectively as shown in the figure below. The Link 'LinkMat' uses these two Variables to construct a matrix and place the result in Variable 'mat'. This Variable is attached to a Performer matrix so it can draw the Virtual hand. Performer has 'Dynamic Coordinate System', DCS, matrices that can change at run time. In DLoVe these matrices change by first attaching Variables to matrices and then modifying the results in them.



The cursor object

The object above is called 'cursor' and when it gets created the user's virtual hand is drawn in the scene.

In the diagram above I am using the polhemus[0] Variable. This array of pointers to polhemus variables is used internally to distinguish the multi-users' hands in a multi-user environment. If the system is running in single user mode, then the Variable polhemus also can be used instead. In multi-user environments however, we need to distinguish the different physical polhemus devices. The index in the array indicates the application level identification number that is supplied by the user in the configuration file. If we have in the configuration file the lines

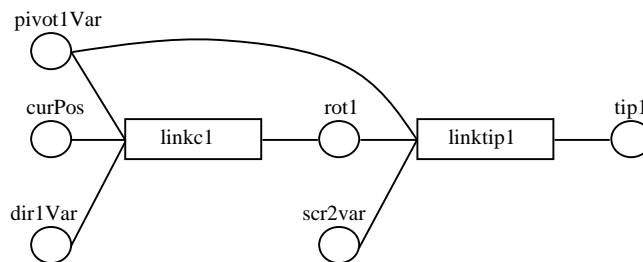
ID: mondian.eecs.tufts.edu=4
ID: vermeer.eecs.tufts.edu=7

then the valid indices in the polhemus array are 4 and 7.

Both users need the following code:

```
cursors[4] = new PolhemusCursor(....., polhemus[4]);  
cursors[7] = new PolhemusCursor(....., polhemus[7]);
```

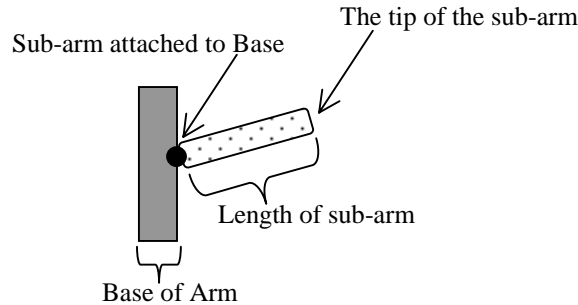
Next I will describe how the arms were designed. All arms are subclasses of the class Arm1. Arm1 arms are single-jointed arms that the user can grab and move. The diagram consisting of the Links and Variables that describes Arm1 arms is shown below:



The Arm1 object

The Variable 'curPos' is the position of the cursor or else, the position of the Virtual hand. The 'pivot1Var' Variable indicates the position of the join of the movable sub-arm and the base. The 'dir1Var' indicates the direction of the arm. The Link 'linkc1' when enabled computes the rotation matrix of the movable sub-arm of Arm1. This is the matrix that gets installed in Performer's scene graph so that when the Variables change the rotation of the sub-arm, the change appears in the scene. Variable 'scr2var' holds the length of the sub-arm. The Link 'linktip1' uses the

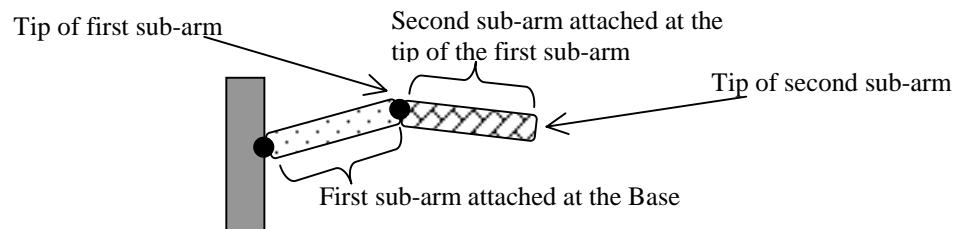
rotation of the sub-arm along with the length of it and calculates the position of the sub-arm's tip.



The parts of an Arm1 object

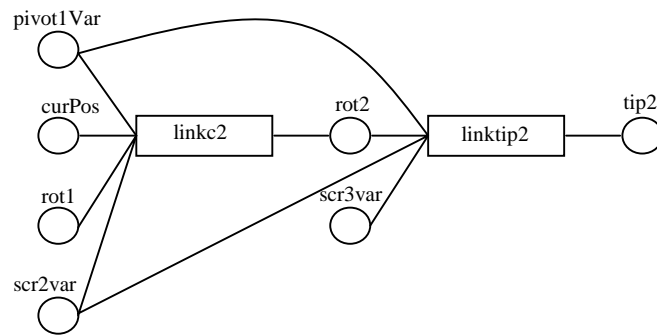
The Link 'linkc1' gets enabled only when the mouse button gets pressed while the arm and the Virtual Hand intersect. This operation indicates that the user grabbed the arm and now can move it. When the user releases the mouse button the hand-arm relationship terminates by disabling the Link 'linkc1'.

Similarly the Arm2 is constructed. Arm2 is a subclass of Arm1. Arm2 is the same as Arm1 in addition to a second sub-arm that is attached at the tip of the first sub-arm as shown below.



The parts of the Arm2 object

In the figure below the second sub-arm (shown below) is similar to the first one shown previously. The second sub-arm needs to also know about the cursor position, Variable 'curPos', and it also needs to know the length of the second sub-arm since it can be of different length than the first one.

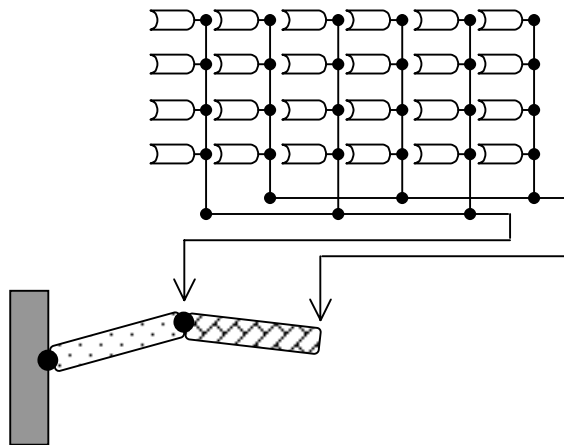


Links and Variables of the Arm2 object

The ArmSlave arm is also a subclass of Arm1. ArmSlave is the same as Arm1 but when created instead of passing the cursor pointer, the tip of another arm is passed. These arms cannot be grabbed. Instead they follow the direction of other arms depending on what object we tell them to follow. The object does not have to be another arm. ArmSlave arms can point to any object's direction, as long as the direction to follow is a Variable of type Variable<Pos6>.

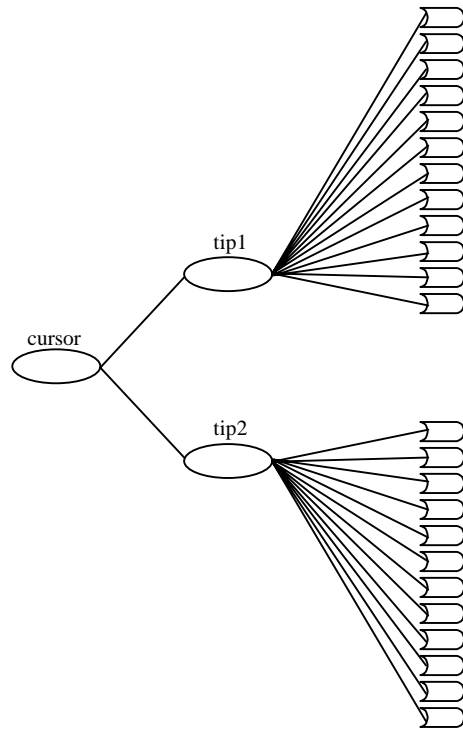
'arms' in distributed mode

If we denote an ArmSlave arm with the following symbol \supset where the output is at the right of the symbol, then in the 'arms' program the wiring looks like the following figure:



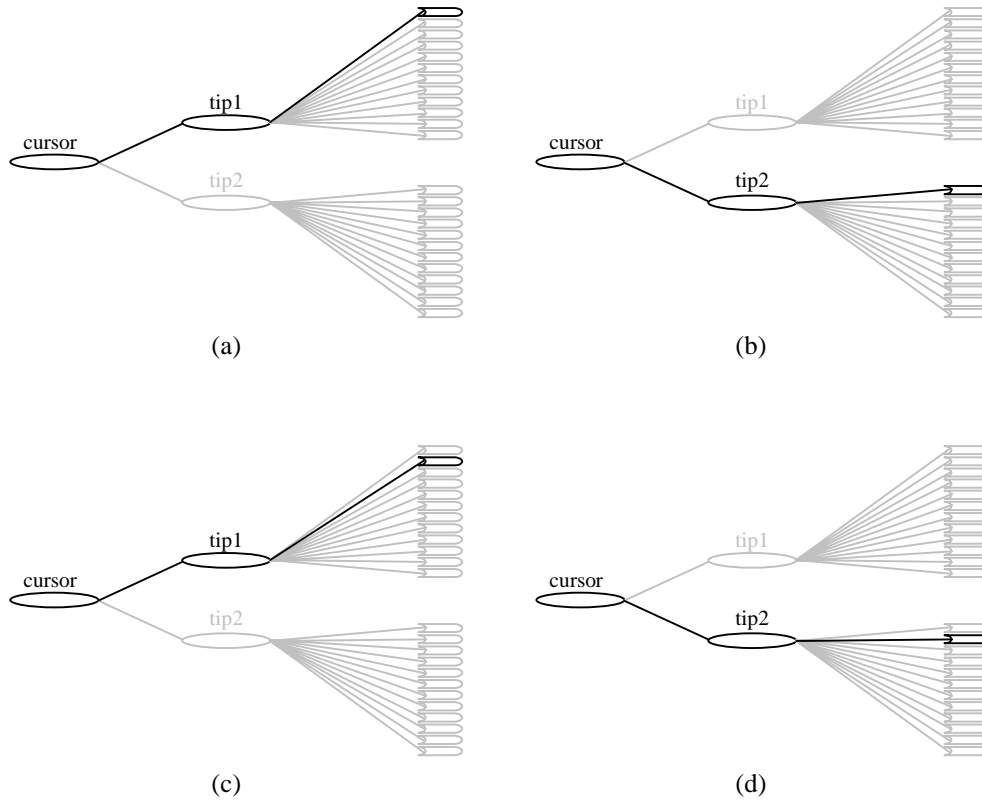
ArmSlave objects wired to an Arm2 object

Half of the ArmSlave arms are pointing at the tip of the first sub-arm of the Arm2 and the other half at the second. A diagram showing the cursor, the two tips of the Arm2 and all the ArmSlave arms is shown below.



Half ArmSlave objects attached to first tip of an Arm2 object and the other half to the second tip

Such an application can run in distributed mode where the position of the ArmSlave arms is calculated in parallel. If we have 24 Workers, then each Worker will calculate the position of one of the twenty-four ArmSlave arms. For simplicity only the first four Workers are shown below along with the ArmSlave arms that are responsible to maintain.



ArmSlave objects evaluated in parallel

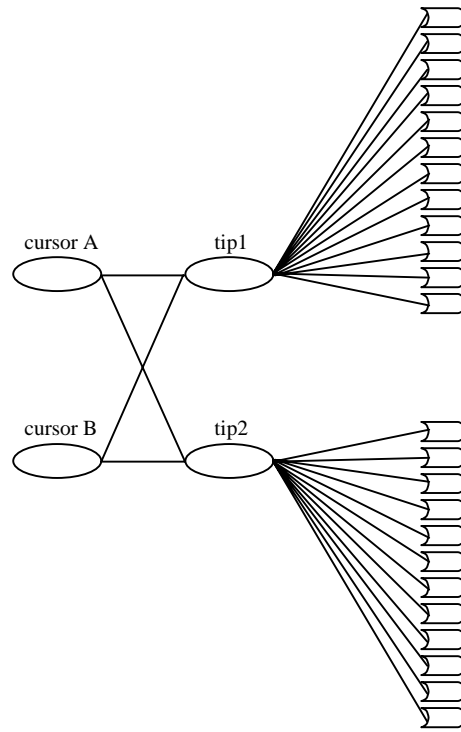
When fewer than 24 Workers are available, Workers will have to maintain multiple ArmSlave arms.

The original program that was designed to run in non-distributed mode can now run in distributed mode where the calculations to maintain the correct direction of the ArmSlave arms is done in parallel. There is no code to modify. The user/programmer only has to recompile and link against the distributed set of libraries of DLoVe.

‘arms’ for a multi-user environment

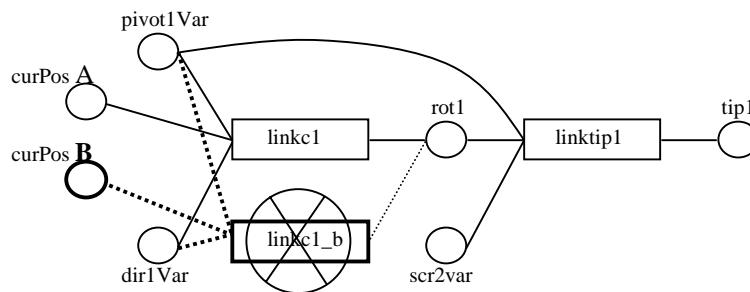
To run the same ‘arms’ program in multi-user multiple users are needed. Multiple users means multiple hands or multiple cursor objects. The number of Slave Coordinators has to be specified in the configuration file. Then the Slaves can start up since the executable file for the Master and the Slaves is the same. In a scenario where two Coordinators exist, where one is the Master and the other is the Slave, both Coordinators control the single Virtual hand. This is because all systems participating in the multi-user environment, Workers and Coordinators, know about a single cursor object. Both Coordinators try to modify the position of the cursor’s position since they are both attached to a Polhemus sensor. The Worker(s) see a SetE operation for the polhemus Variable. Clearly we need two cursor objects. One for the Master and one for the Slave Coordinator since both users need a Virtual hand in the Virtual World.

We also need to add some additional Links and Variables and wire them such that both cursors/users can grab the sub-arms of Arm2. A diagram with the additional cursor and the additional wiring is shown below:



Two-user interface in the 'arms' program

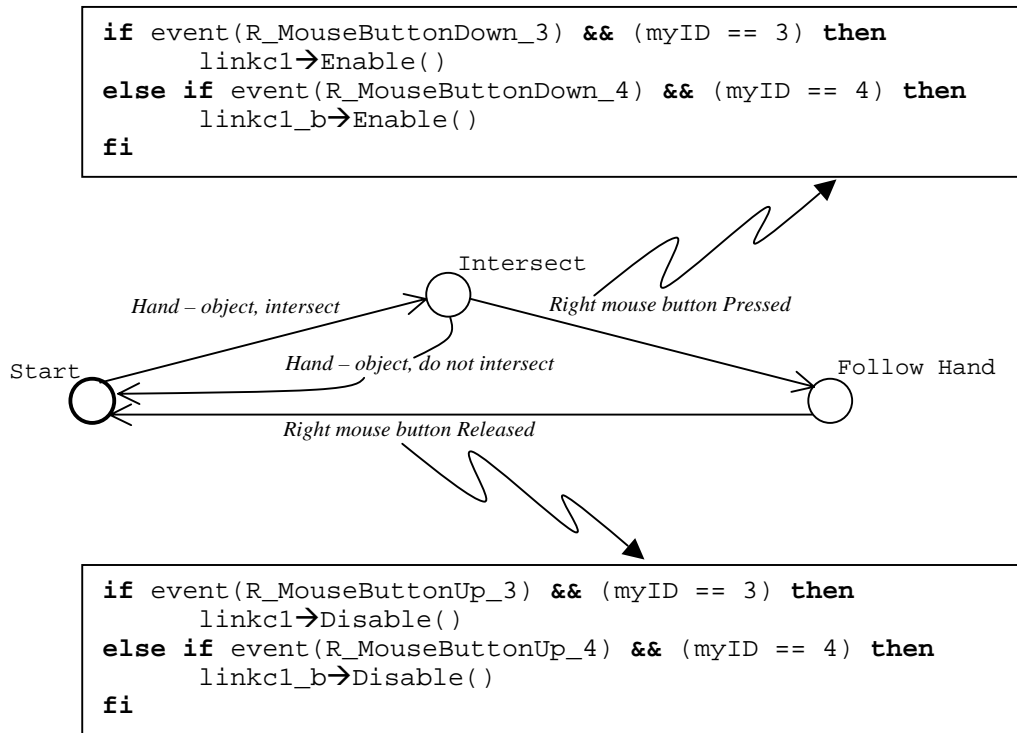
First the new cursor, cursor B above, must be created. Then all the Links that are using cursor A as input must be duplicated so cursor B can be attached to these new Links as shown below highlighted:



Only one user controlling an Arm2 object

Link 'linkc1' uses cursor A for input and so it must be duplicated. The duplicated Link is 'linkc1_b'. When creating 'linkc1_b' the cursor B has to be passed instead of cursor A. The rest stays the same. The important point here is to note that the output Variable of the duplicated Link is the same. This indicates that both cursor can modify the position of the sub-arms. The same procedure needs to be done for any other object in a program that gets transformed from a single-user to multi-user program.

The second part of the modification is the state machine. The sub-arms can be moved only when the user intersects his/her hand with a sub-arm and presses the mouse button, which in turn enables the 'linkc1'/'linkc1_b' Link. The discrete time machine sends events for every mouse event it catches. When the mouse button gets pressed, the discrete time machine will send an event indicating that the mouse button was pressed to all objects in the system. And thus, both Links 'linkc1' and 'linkc1_b' will get the event and they will both get enabled, eventhough only one user pressed the mouse button. For this reason the discrete time machine also re-writes the label of the event appending the application identification number at the end of each event, which is taken from the configuration file. The programmer needs to modify the code where the event gets caught in the Links. Each Coordinator needs to compare its application identification number with the received event to determine whether it needs to enable its Link or not. They both need to compare the events against the application id because they both run the same executable and the only way to figure out who is who, is by checking their ids. In this environment having one Coordinator with id equal to 3 and the other equal to 4 the state diagram looks like the following figure:



State transition diagram of the Event Handle of an Arm1 object and the code that is executed based on the state transition

The code that gets executed is shown the boxes. The Coordinators need to test if the event gets generated locally so they can perform the correct operation on the correct Link. The Workers do not execute the code in the boxes. The Coordinators only execute it. The Coordinators generate the events and then based on the event they perform an operation, in this case Enable and Disable operations. Then these Enable/Disable operations are sent to the Workers over the network as requests to modify the graph accordingly.

With the above state diagram, both users can grab the sub-arm at the same time. And both 'linkc1' and 'linkc1_b' Links can be enabled at the same time. Because both Links store their output in the same Variable 'rot1', half of the time one user

controls the arm and half the other. To work around such problems more conditional code needs to be inserted between the 'if' statements above to ensure that the first user who grabs the sub-arm controls it until the user releases the mouse button.

The 'switch' mechanism'

Transforming a program from single-user to multi-user involves modification of existing code. Adding functionality however involves adding new code. In multi-user environments it makes sense to give different roles to different users. For example, in a virtual operating room there is one instructor that is teaching new graduates how an operation is performed, and the graduate students that are watching and learning. Or in our example of the double-jointed arm and the ArmSlave arms, one user is allowed to grab only the first sub-arm and the other the second sub-arm. This can be specified with the diagram above. However, how can the roles be switched at run time. Let us say that user A is allowed to grab the first sub-arm and user B the second. Then by pressing the character 't' the roles get swapped. Or by pressing the character 'T' both users are allowed to grab both sub-arms. The difficulty of the problem is that both users can type in the character 't' or 'T'. How would one user know that the other user typed the character 't'? If we have a single Variable 'V' that both Coordinators can query we might drop the message that includes a GetE operation on Variable 'V'. Multi-messaging mechanism without blocking for replies tries not to overflow the network by dropping messages. Querying Variable 'V' in every loop in the main program is not a solution because events are happening in discrete not in continuous time.

If we have two users, user A and user B, where user A initially is allowed to grab arm A and user B arm B and then user A presses the 't', both users might end up grabbing arm B at the same time. When user B finally sees the change, he/she will switch roles but it may be too late and user A will be confused because of the wrong behavior of the application. In the following figure the action of the users is shown and also the value of the Variable 'V' as seen by each Coordinator and assuming just one Worker. When the Variable 'V' is set to 0 then user A can grab the first sub-arm and the user B the second. When the Variable 'V' is set to 1 the roles are reversed.

	User A	User B	Worker
(a)	0 Grab A	0 Grab B	0
(b)	1 Press 't'	0 Grab B	1
(c)	1 Grab B	0 Grab B	1
(d)	1 Grab B	0 Grab B	1
(e)	1 Grab B	0 Grab B	1
(f)	1 Grab B	1 Grab A	1

Inconsistency issue in a multi-user VE

Initially (a) user A is allowed to grab arm A and user B arm B. All see 'V' as set to 0. Then user A types in the character 't' (b) to switch roles with user B, and sets 'V' equal to 1. User A sends a message to the Worker and both Coordinator A and the Worker see 'V' as equal to 1. However, Coordinator B does not see the change. This is because Coordinator B dropped the message that included a GetE request on Variable 'V'. To guaranty that this SetE operation arrives at the Worker we also

have to send the message as a high priority message so it does not get dropped. This can be implemented by Enabling a fake Link; Enable and Disable operations are of high priority. In (c) and (d) both users can grab arm B. The same happens in (e) as well, but now the Coordinator actually sends the message to the Worker with a GetE request on Variable 'V'. Finally in (f) the Worker replies back to Coordinator B and all machines see the same Value on Variable 'V'. It is more confusing when the Coordinator B still thinks that Variable 'V' is set to 0 and types in 't'. This mess is due to the fact that we are using the continuous time subsystem to implement something that is discrete. The Coordinators are querying a continuous Variable that it changes based on discrete events. That is why we need a message that can travel from a Coordinator to all other Coordinators. But since there is no direct connection between the Coordinators, this can be done via a Worker. Using the "Backward Notification of Events", a Coordinator can request a Worker to notify all other Coordinators. This way the Coordinators do not have to query in every loop that special Variable 'V'. When one Coordinator wants to set such a special Variable it uses the SetE_Broadcast() operation which is also of high priority message. The Worker sets the value on the requested Variable and then it sends a message to all other Coordinators to notify them about the change. This message looks like a GetE reply from the Worker to the Coordinators. The Coordinators do not care if they requested that Variable or not. This way a change on one Coordinator is visible by all other Coordinators in the multi-user environment.

Chapter 11: An Eye Tracking Application

Overview

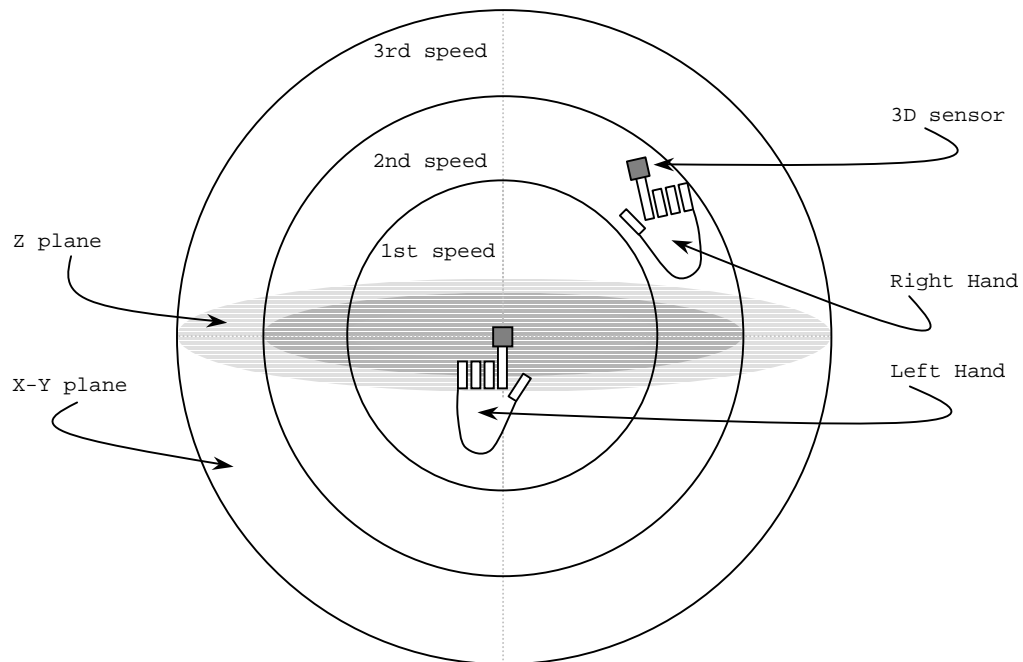
DLoVe is designed to adapt to the evolving needs of non-WIMP user interfaces. DLoVe supports the mechanisms for new input and output devices without requiring modifications to the UIDL. In this chapter I present an application that reads input from an X window, a Polhemus 3D tracker for the location and the orientation of the head and hands, and from an eye-tracking device.

In this application, 2Zoom, there is a virtual world with virtual objects away from the reach of the user. When the user looks at a virtual object for over 5 seconds, the object becomes selected and highlighted. The user can operate on the selected object using his/her hands. To deselect an object, the user has to look away from the selected object. All the virtual objects are semi-transparent. The more the user

looks at an object, the more solid it becomes up to the point when the object becomes selected. The left hand of the user simulates the origin of the operation, and the right hand operates on the object. The right hand, depending on the location of the left hand, can move and rotate the remote virtual object in space. When the right hand comes close (within an inch) with the left hand, the object stops moving.

Hand Movement for Object Manipulation

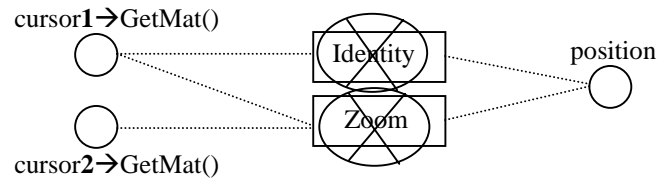
The user using both of his/her hands can move and rotate selected objects from a distance. When the object is near his/her reach, he/she can reach out and grab the object and manipulate it. In contrast, when an object is far away the user has to bring the object near him/her to examine it or build more complex objects out of more primitive objects. The left hand serves as a base or origin that the right hand uses as reference. The right hand can move around the left hand to indicate in which direction the selected object should move. There are three perimeters around the left hand, which represent the speed of the remote object. For example, when the right hand moves on top of the left hand, the remote object keeps moving upward. When the right hand moves about 15 inches on top of the left hand, the remote virtual object still keeps moving upward but with a higher speed. To slow down the object the user has to move his right hand closer to the left. When the right hand comes to an inch away from the left, the object stops moving. The following figure illustrates the movement of the hands to manipulate a remote virtual object:



Hand movement for eye-selected object manipulation

Designing the Application

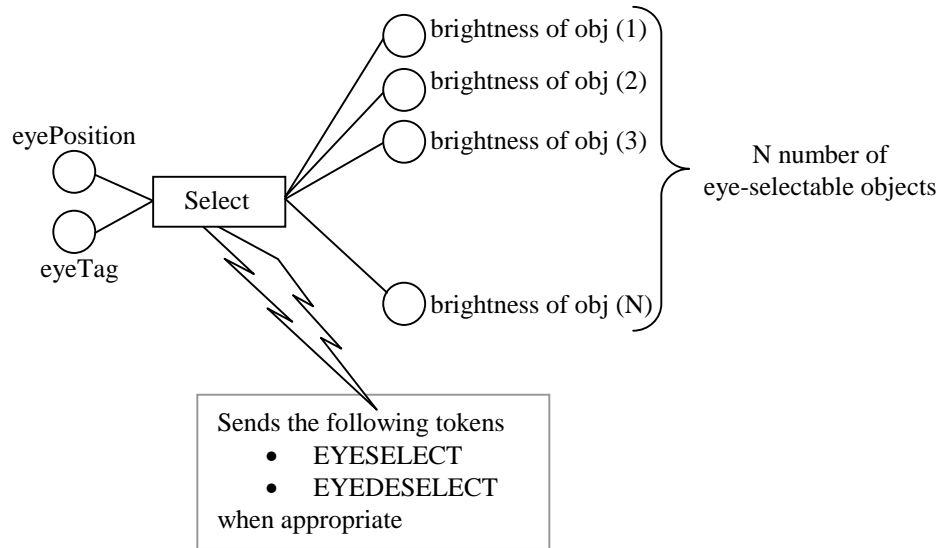
Each object can be selected by looking at it for over 5 seconds and then manipulated, or if it is close enough to the user, it can be grabbed and then manipulated. The following figure shows the Links and Variables that describe each object this application:



Two hand positions controlling the position of an eye-selected object

Each object is encapsulated within a 'TargetSelect' class. Each object, if it is close to the user, can be grabbed with one hand in which case Link 'Identity' becomes enabled and Link 'Zoom' becomes disabled. In this scenario, the object follows the position of 'cursor1', which is the user's right hand. If the object is selected, (e.g. by looking at it for over 5 seconds) it can be manipulated using both hands.

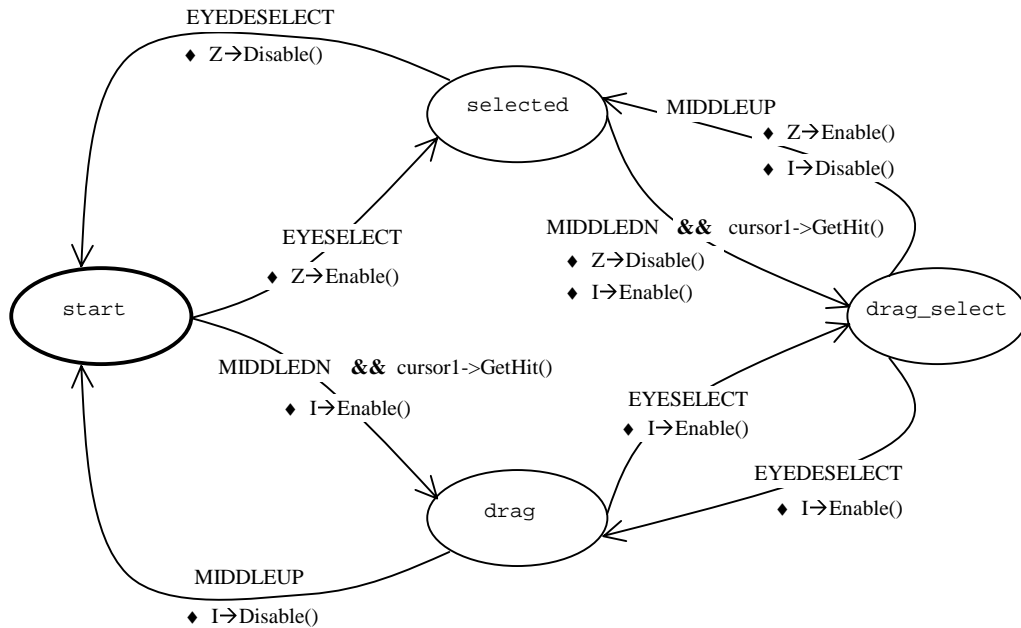
There is also one Link in the application that controls the brightness and the selection of the objects. The 'Select' Link uses the 'eyePosition' and 'eyeTag' Variables that indicate in which direction the user is looking. The 'Select' Link keeps a histogram of the 'brightness' value of each object. If the user looks at an object for over 5 seconds, the Link initiates a token to that object's event handler that indicates that a specific object is selected.



Brightness and eye-selection of objects

When an object becomes selected, the Link initiates an `EYESELECT` token to the selected object's event handler. And when the object transitions from being selected to not being selected, an `EYEDESELECT` token is initiated.

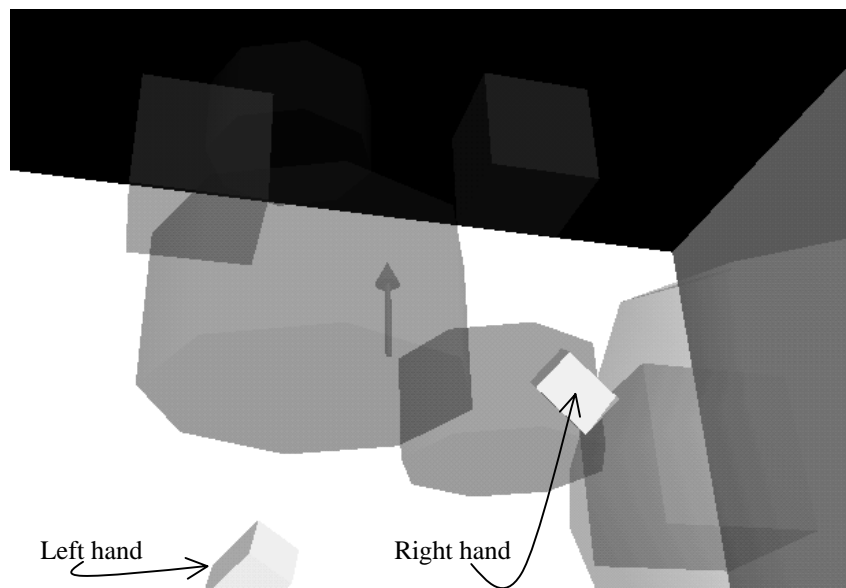
The event handler of each object consists of four states. Initially all objects are in the 'start' state by default. In the following figure, the tokens that cause the state transition are shown on top of the arrows and the lines with the ♦ indicate the action that takes place during the transition; which is enabling or disabling Links. The 'z' pointer refers to the 'Zoom' Link and the 'I' refers to the 'Identity' Link. Some state transitions involve a test condition to figure out whether or not the user's hand intersects with an object (e.g. from 'start' to 'drag', and from 'selected' to 'drag_select').



State transition diagram of the 2Zoom application

For example, if the user looks at an object for over 5 seconds, the 'Select' initiates an EYESELECT token, the object's event handler enables Link 'Zoom' and then it transitions to the 'selected' state; where the object can now be manipulated using both hands. If the user moves the object near his/her reach and grabs it (by intersecting his/her hand with the object and pressing the middle mouse button), DLoVe initiates a MIDDLEDN token. The event handler sees the token, checks to see if the user actually touches the object, and if it does it disables Link 'Zoom', enables 'Identity', and transitions to state 'drag_select'. At this point the object follows the user's hand position and orientation.

A snapshot of the application at run time is shown below:



Snapshot of the 2Zoom application

The two hands of the user are visible as well as several objects that can be selected and manipulated.

Chapter 12: The DLoVe Virtual Park

Introduction

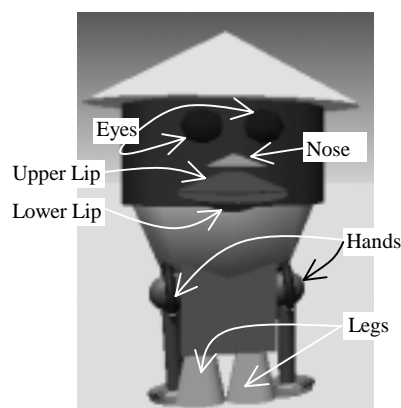
The programming examples given in Chapters 10 and 11 were each designed to test a limited subset of the DLoVe system's purported features and capabilities. These small applications echo the themes that large-scale non-WIMP interfaces encounter. However, these examples did not require facing challenges encountered in full scale Virtual Environment application development. Developing a large application using DLoVe's paradigm can claim its programming suitability for Virtual Environments. The application developed in this chapter was done on a Silicon Graphics workstation, using Performer as the graphics-rendering library.

Three variations on the basic design were implemented. The first is a desktop application that allows the user to explore and interact with the world; using keyboard and mouse controls. The second is a head mounted display version that employed Polhemus trackers to sense the user's position, orientation, perspective,

and activities within the virtual environment. The third is a multi-user implementation where one user uses the keyboard and mouse, and the other the Polhemus and an HMD to interact.

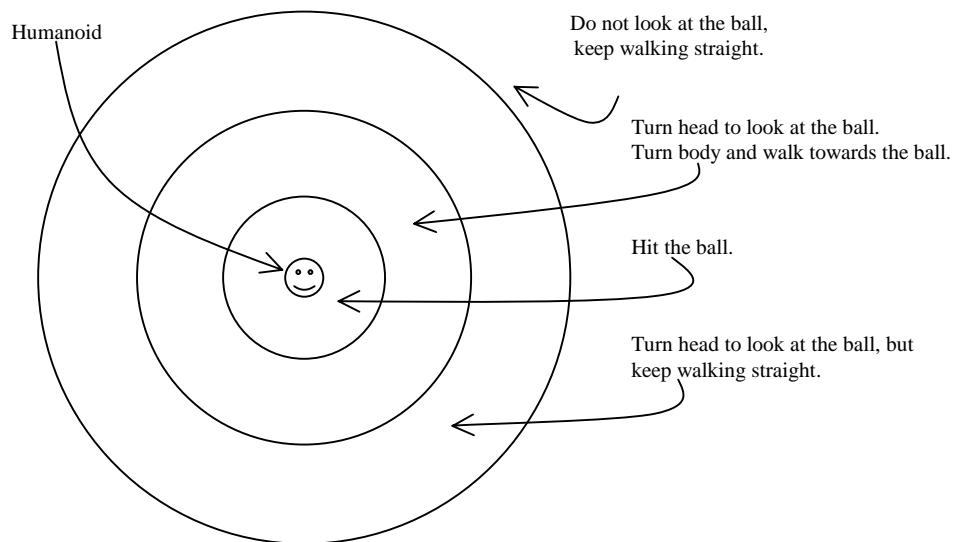
The Actors

The application consists mainly of a Virtual Park with trees, computer simulated entities called Humanoids, and several plane-like objects that are flying over the park. The main actors in the park are the Humanoids. The Humanoids are entities that walk and interact with each other and with a virtual ball, or just wonder in the park. In their spare time they play with a virtual ball. They also have a constraint that makes them always stay in the playground and not go on the green. Both, the users, and the Humanoids, can hit the ball and change its velocity and trajectory. A Humanoid is shown below:



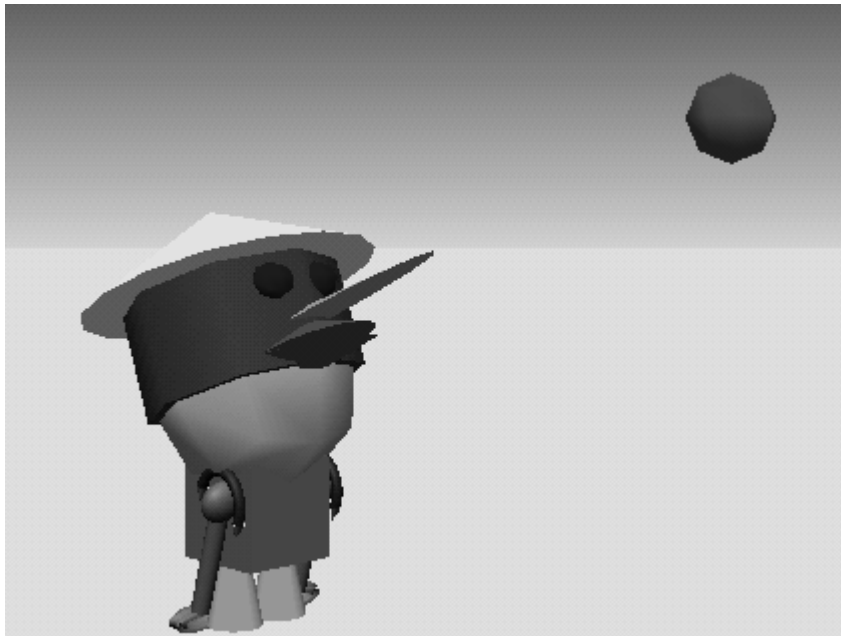
A Humanoid and its parts

The Humanoids have eyes and ears, and so they can hear and see the ball and each other. For example, if they are walking straight and the ball drops just behind them, they rotate so that they face the ball and eventually hit it. Their heads, when the ball is close enough, follow the trajectory of the ball. The Humanoids do not always follow the ball. If they are very close to the ball, they rotate their body to walk towards the ball. If the ball passes by them, then they only look at it and they keep walking the same direction they were. But even in this case, they cannot rotate their heads over 45 degrees. When they can reach the ball, they lift their hands and hit the ball at which point the ball starts traveling upward, up to a point where gravity will bring it back down to the park. The following figure shows the actions taken by a Humanoid depending on where the ball is:



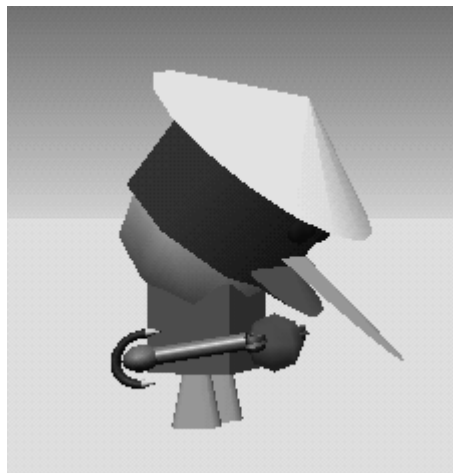
Actions of a Humanoid based on the Virtual ball's position

The following snapshot shows a Humanoid looking at the traveling ball in the virtual park:



A snapshot of a Humanoid that its head follows the trajectory of the ball

The following snapshot shows a Humanoid at the time when he hits the ball. The Humanoid has just lifted his hand to change the velocity and trajectory of the ball.



A Humanoid at the point of hitting the ball

The Humanoids can be picked up and moved by the user(s). They do not like the experience, however, and they go into a panicking mode where they move their heads left and right, their hands and legs back and forth, and their lower lip up and down.

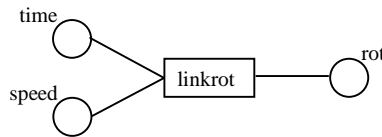
There are also other objects in the park such as sliders and arms. The user(s) can manipulate the sliders to modify the speed of the flying objects, and the arms to modify the position of the lower lip of the Humanoids. All Humanoids know where all other Humanoids are in the park and they try to avoid them by not colliding into them. The arms objects were explained in chapter 10 and the sliders with the rotating objects will be explained below.

When this application is executed in a two-user environment, each user has a different roll, and they can also see each other. For example, one of the users is allowed to only grab the first arm of the 2-handle arm, and the other user the second. However, the users can reverse the roles by pressing the character 's' on the keyboard. When the 'b' character is pressed, then both users can grab both of the arms of the two-handle arm. Both users can type the characters 's' and 'b' to reverse their roles.

Sliders and Orbiting objects

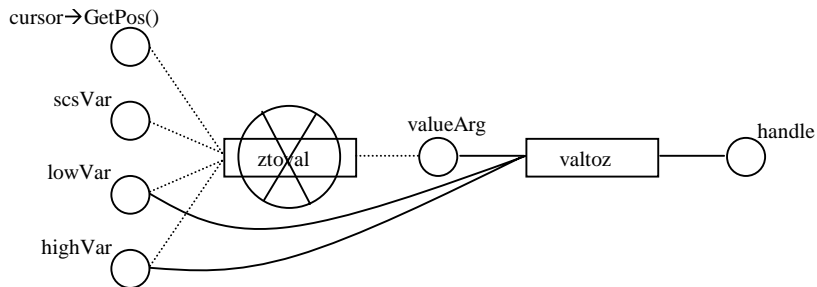
The sliders are simple objects that have a handle bar that can be moved up and down to change a value of a Variable. The orbiting objects depend on time and

speed. By modifying the value of the slider we can control the speed of the orbiting objects. The following diagram shows the Links and Variables that make up an orbiting object:



An orbiting object

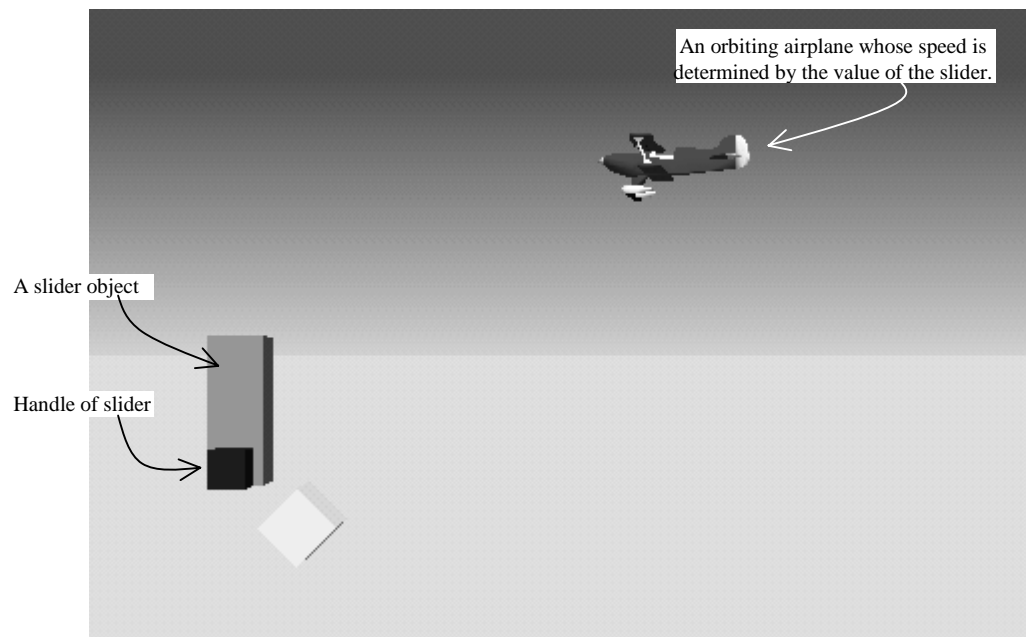
An orbiting object uses the output Variable 'rot', which is a performer matrix, to draw itself based on time and speed. The user can modify the speed of an orbiting object using the slider. The Links and Variables diagram that make up the slider object is shown below:



The Slider object

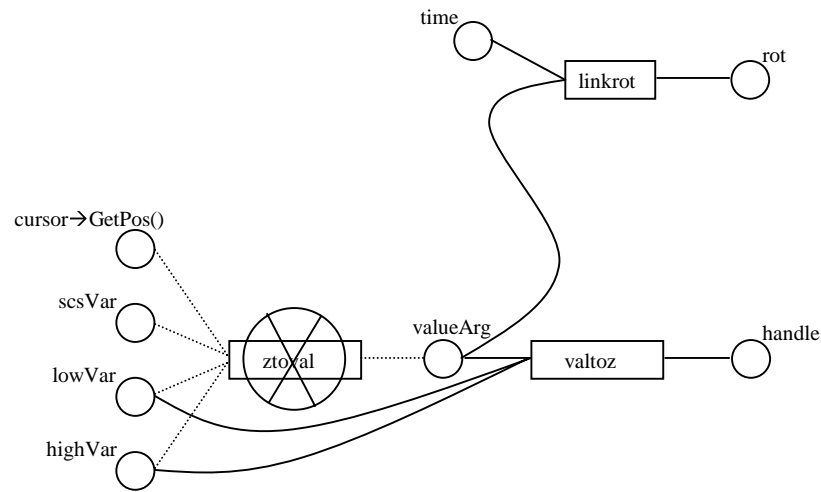
The 'lowVar' and 'highVar' Variables indicate the minimum and maximum values of the slider. The Variable 'valueArg' indicates the value of the Slider, and the 'handle' Variable indicates the position of the handle of the slider. The user's hand position is the Variable 'cursor->GetPos()'. When the user grabs the handle of the

slider, Link 'ztoval' becomes enabled and the user can modify the position of the handle and as a result the value of the slider (Variable 'valueArg'). A slider object that controls the speed of an orbiting object is shown below:



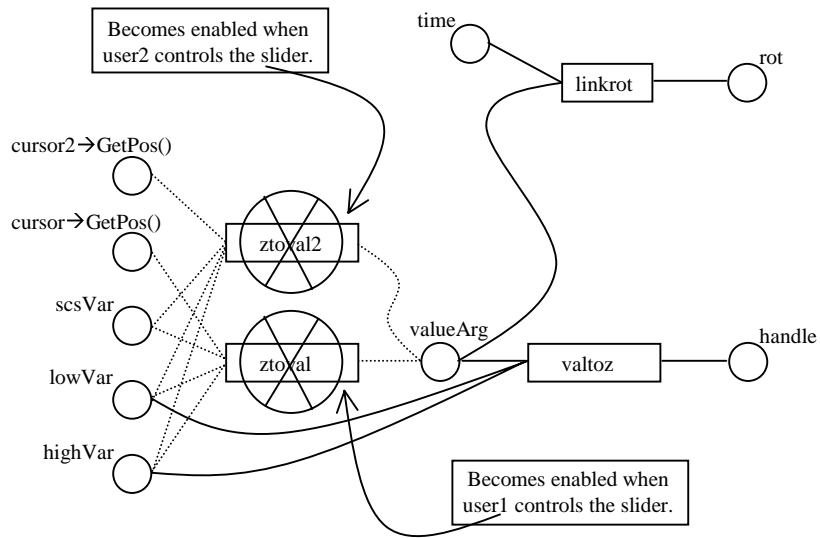
Snapshot of a Slider controlling an orbiting object

Variable 'valueArg' is used as input to the orbiting object to modify its speed. The following diagram shows the wiring of a slider and an orbiting object where the slider can modify the speed of an orbiting object:



Slider modifies speed of an orbiting object

As with the 'arms' program, when the slider object is used in a two-user environment, Link 'ztoval' needs to be duplicated since it is using a Variable that is attached to the user's hand (in this case `cursor->GetPos()`). For example, in a two-user environment, the above diagram looks like the following figure:



Slider wiring in a two-user environment

In the figure above, each user's hand is attached to the Variables 'cursor->GetPos()' and 'cursor2->GetPos()' respectively. The first user's hand is attached to Link 'ztoval' and the second user's hand to 'ztoval2'. Each user can only enable the corresponding Link, and as a result, each user can control the corresponding Link independently from each other.

Collision Prevention

Each Humanoid has a unique id and based on this id and the force, explained below, the Humanoids prevent collision with each other. Each Humanoid knows the position of all the other Humanoids. All Humanoids act as magnets with the same

polarity that makes them push each other away based on a force that is calculated as follows:

```
1 CalculateForce()
2     force = 0;
3     MyPos = thisHumanoid->GetPosition();
4
5     for( i = 0; i < TotalHumanoids; i++ ) {
6         for( k = 0; k < TotalHumanoids; k++) {
7             foreach Humanoid man do
8                 other = man->GetPosition();
9                 if( InRadius( MyPos, other, 25) ) {
10                    if ( k > 2*i || i > 3*j ) {
11                        force += (i + k);
```

Not all Humanoids affect each other. Only the ones that are near (within a radius of 25 -line 9-) to the Humanoid in question are in consideration. Out of these Humanoids in consideration, only the ones that actually satisfy the condition in line 10 above affect the Humanoid in question.

When a Humanoid with a low id is going to collide with a Humanoid with higher id, it gets pushed back based on the force calculated as shown above, and then it stops moving for a few seconds. The other Humanoid, the humanoid with a higher id, also gets pushed back based on the force but it first changes its rotation and then keeps moving.

Implementation Issues

Implementing the collision prevention algorithm for a single/non-distributed environment was straightforward. However, several issues were encountered when an attempt was made to execute the same program in the distributed mode. In the

non-distributed mode, each Humanoid before updating its position, it is doing a collision check to prevent collision with the other Humanoids. However, in a distributed environment where each Humanoid is simulated by a different Worker, the current design does not work. This is because each Worker only knows about the position of the Humanoid it is simulating and no one else's. To prevent collisions, a Worker needs to simulate all Humanoids. However, there is an application design strategy that can be implemented, where a Worker knows the position of all other Humanoids that are being simulated by other Workers, even though each Humanoid is simulating a subset of the total number of Humanoids. This can be implemented using 'synthetic Variables'.

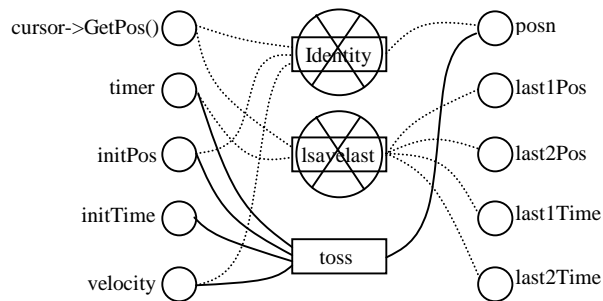
Synthetic Variables are regular Variables in DLoVe that they get updated by the application and not by DLoVe's constraint engine. The Coordinator collects all the Humanoid positions from all the Workers. Then it updates all the Workers with all the Humanoid positions so they can run their algorithm for collision prevention. A detailed explanation of this implementation is given below.

Designing the Virtual Park

The Humanoids are the main actors in the Virtual Park. Their behavior is primarily determined by the virtual ball and by the position of most of the other Humanoids. As explained previously, the Humanoids try to find where the ball is so they can hit it. However, if they collide with other Humanoids they get pushed back and maybe away from the ball.

The Virtual Ball

The ball is an object that can be hit by the Humanoids and travel over the park. Gravitational forces bring it down to the park. The user(s) can also grab the ball and toss it in the park. The Links and Variables with which the ball is made up, is shown below:



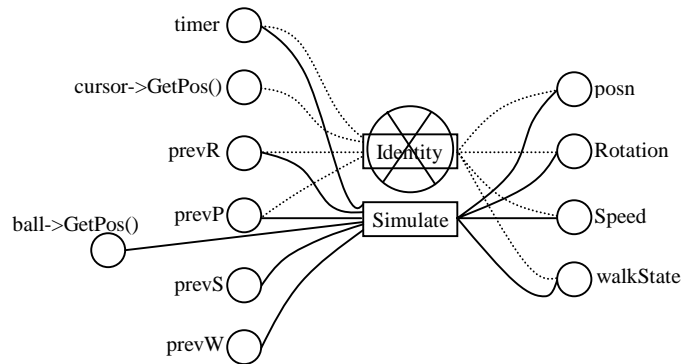
The virtual ball

The Links 'Identity' and 'lsavelast' become enabled when the user grabs and moves the ball. Otherwise, Link 'toss' uses the last position of itself, the time, and the velocity, to determine its position at a given time.

The Humanoids

The Humanoids are computer-generated entities that walk and wander around the virtual park. They can be picked up by the user and manipulated. Their main

interest is to find the virtual ball and hit it. The following diagram shows the Links and Variables and their connections that make up a Humanoid object.



A Humanoid object

Each Humanoid knows where the ball is located, so it can take the appropriate action depending on its position. Link 'Simulate' also runs a collision prevention algorithm that prevents Humanoids colliding with each other. When the Virtual Park application is executed in a single non-distributed environment, each Humanoid can figure out where all other Humanoids are by executing the following code:

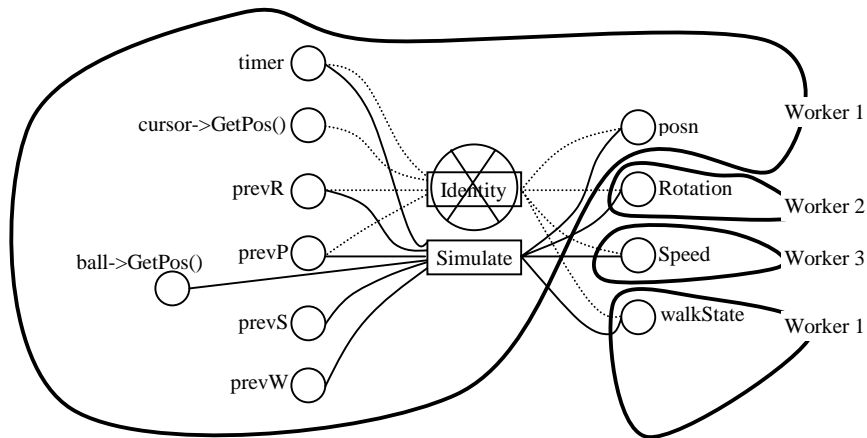
```
foreach Humanoid i; do
    i->GetE();
done
```

However, in a distributed environment collision prevention will not work with the current design because the Workers simulating one Humanoid cannot determine the position of the other Humanoids that are being simulated by the other Workers. They would have to simulate all Humanoids, and so, there would not be a need for

multiple Workers. Synthetic Variables will play a key design strategy in solving this problem, and the implementation is shown later.

Partition Issues

With the current design of the Humanoids, DLoVe would partition the constraint graph very inefficiently. The partition algorithm, as explained in chapter 7, assigns Variables to Workers based upon the number of Links that a Variable depends. In the following figure where 3 Workers are available, the partition algorithm may assign 'posn' and 'walkState' to Worker 1, 'Rotation' to Worker 2, and 'Speed' to Worker 3.

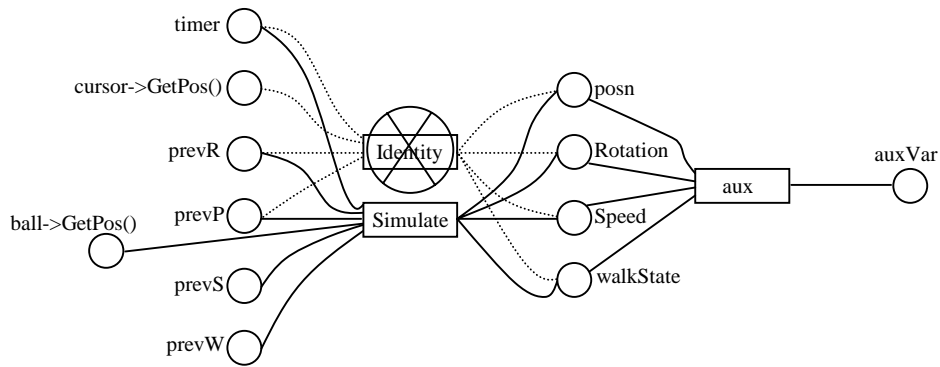


Issues in partitioning a Humanoid's graph

This above assignment is done as shown above because each of the four output Variables depends upon the same number of Links. Even though the algorithm correctly assigns the output Variables to different Workers, it makes sense to have a single Worker updating all four Variables of the same Humanoid; all four Variables

become updated by a single evaluation of the Link 'Simulate' or 'Identity'. In addition, Workers 2 and 3 would also spend time evaluating the Links involved to bring these Variables up-to-date, ending up with redundant evaluations. A single evaluation of the Link 'Simulate' brings all four output Variables up to date.

To fix this inefficiency, I attached all four output Variables to an auxiliary Link 'aux' that is using an auxiliary Variable 'auxVar' as output, as shown below

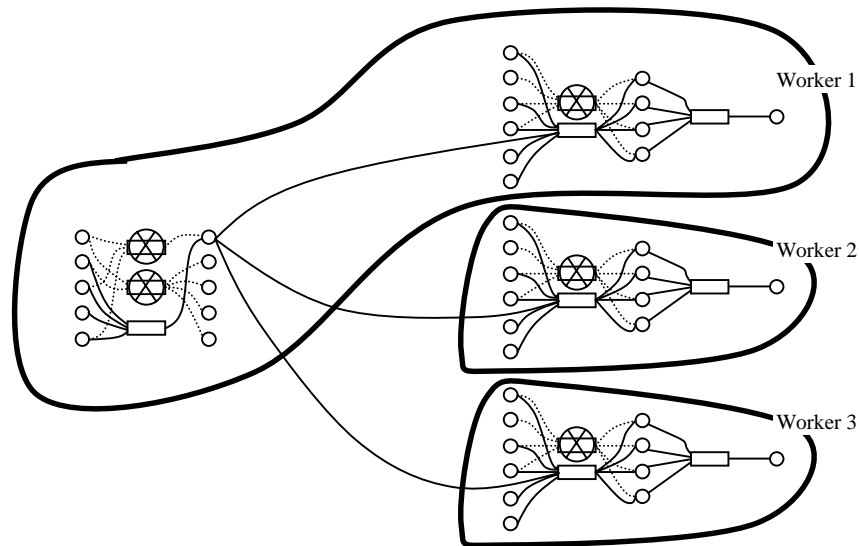


Correcting the partition inefficiency

The Link 'aux' and the Variable 'auxVar' are never used by the application at run time. They are only there to give a hint to the partition algorithm to assign all four output Variables of each Humanoid to a single Worker.

Collision Prevention and ball simulation issues in a Distributed Environment

In a non-distributed environment each Humanoid can determine the position of all other Humanoids very easily as explained previously. In a distributed environment however, where each Humanoid is simulated by a different Worker, there is no way for a Humanoid to know the position of the other Humanoids. This is because, by design, there is no direct communication between the Workers. The following figure illustrates how three Humanoids are attached to the ball and how they are partitioned so that, they are simulated by a different Worker.



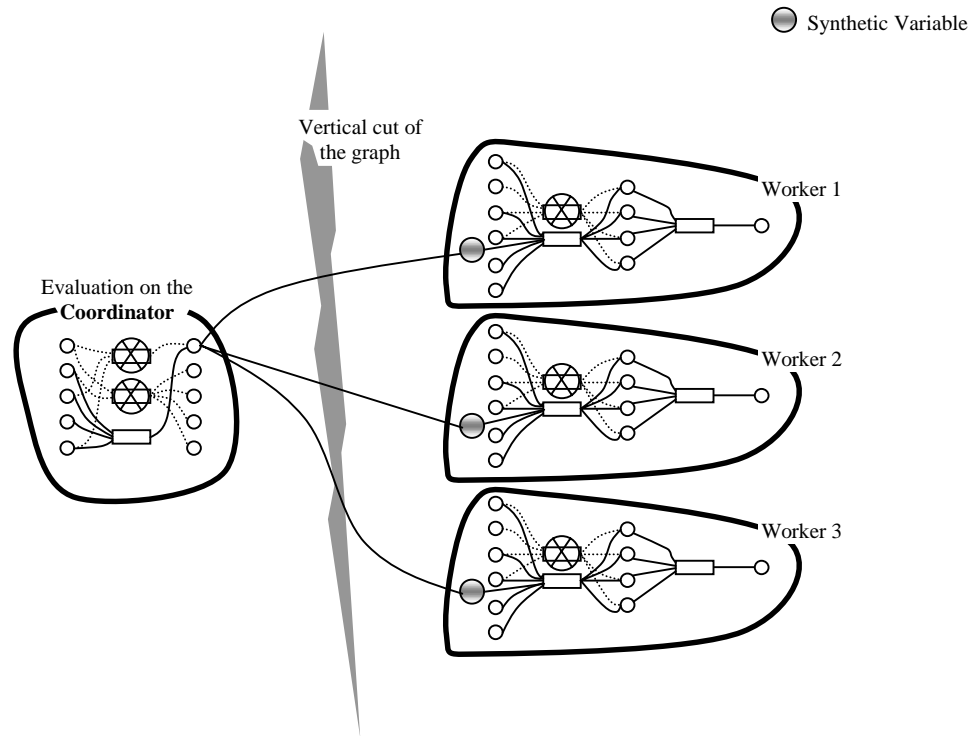
Three Humanoids assigned to be simulated by three different Workers

In addition to this problem, in the figure above each Humanoid also simulates the position of the ball. Since each Humanoid only knows about its position, it will not

know when some other Humanoid hit the ball. As a result, each Humanoid will think that the ball is at a different position.

Synthetic Variable for ball position

To fix the problem that each Humanoid simulates its own version of the ball, the Humanoids are not wired to the ball (so they do not update its position) but still have to know the position of the ball. To solve this problem I use a synthetic Variable. Each Worker has a synthetic Variable as input, which indicates the position of the ball. The Coordinator, which simulates the trajectory of the ball, informs all Workers about the position of the ball. This way all Humanoids have a consistent value of the position of the ball and they can act accordingly. The Coordinator, which simulates the ball, and also knows the position of each Humanoid, changes the trajectory of the ball and eventually informs all Workers. The partition algorithm cuts the graph horizontally, where with the use of synthetic Variables the designer cuts the graph vertically; something that is left for future work to be automated. The following figure shows the use of synthetic Variables for the position of the ball:



Synthetic Variables for the position of the ball

In the figure above, each Humanoid is simulated by a different Worker. The Coordinator simulates the position of the ball. The Humanoids are not directly attached to the position of the ball. However, they know the position of the ball via a synthetic Variable. The Coordinator that updates the position of the ball, informs all the Workers about the position of the ball, which is stored in a synthetic variable.

The Coordinator executes the following code in every loop:

```

forever do;
    bpos = ball->GetE();
    for every Humanoid i; do
        i->GetBallSyntheticVar()->SetE(bpos);
    done;
done;

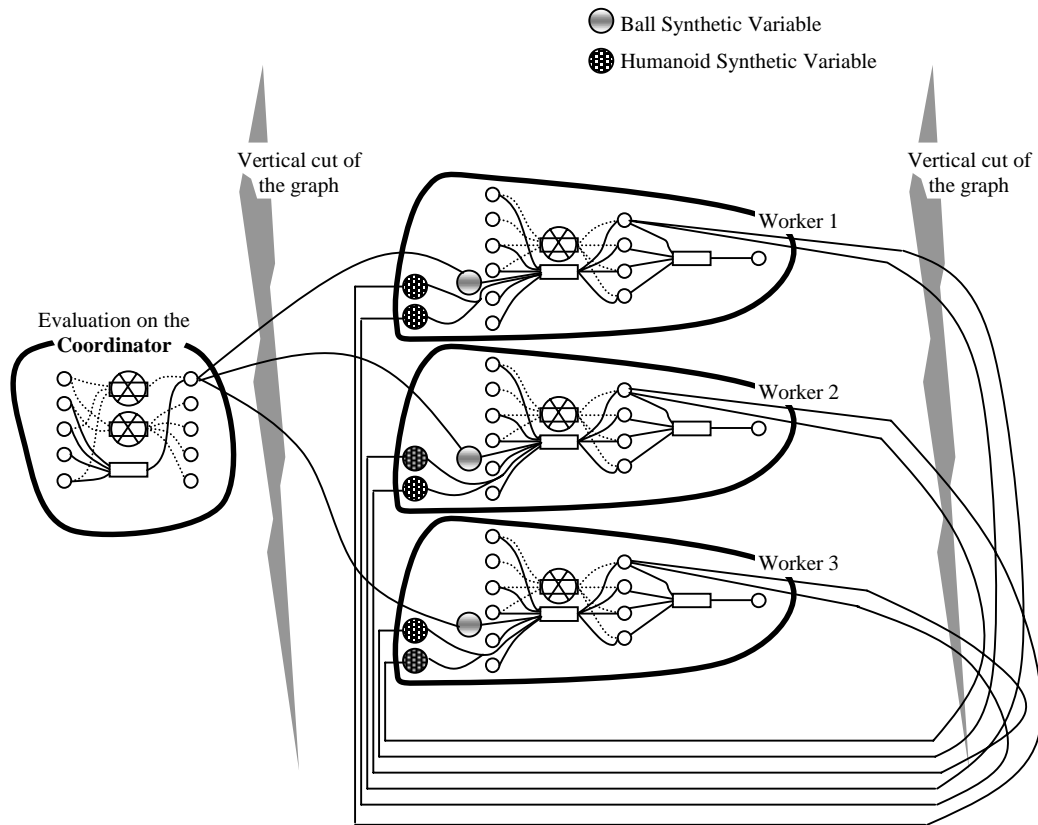
```

This way all Humanoids know the consistent position of the ball and they act accordingly.

This is a solution to this problem that the designer needs to make. DLoVe partitions the graph based on a horizontal cut where each Variable is assigned to a Worker based upon the number of Links it depends. The use of synthetic Variables requires the designer to cut the graph vertically, at the design level, and insert code in the application that updates all the synthetic Variables in the system.

Synthetic Variables for Collision Prevention

The Humanoids needs to know about all other Humanoids' position to prevent collision with each other. But, all Humanoids are simulated on different Workers. To solve this problem I used synthetic Variables, as I did for the ball's position. The Coordinator, which queries all Workers, knows all the information about all the Humanoids so it can draw them on the screen. The Coordinator queries every single Worker about the position of the Humanoids and then informs all the Workers about all the other Humanoids. The following diagram shows the use of synthetic Variables for every Humanoid to know the position of every other Humanoid:



Use of Synthetic Variable for Collision Detection

The Coordinator executes the following code to update the ball's position and also the position of every Humanoid:

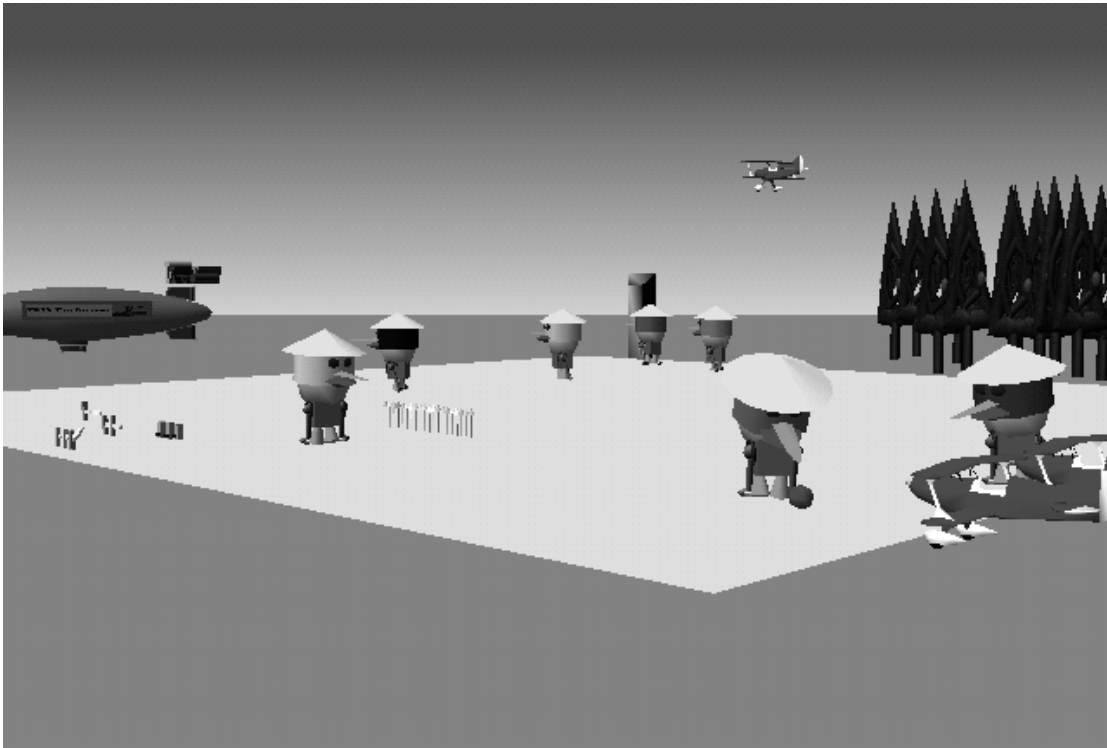

```
1 foreach iteration;
2   bpos = ball->GetE();      // get the ball's position
3
4   //
5   // get position of each Humanoid
6   //
7   for every Humanoid i; do
8     i->GetPos()->GetE();    //
9   done;
10
11  //
12  // update synthetic Variable of balls position
13  // and Humanoid's position
14  //
15  for every Humanoid i; do
16    i->GetBallSyntheticVar()->SetE(bpos);
17
18    //
19    // inform every Humanoid about every other Humanoid
20    //
21    for every Humanoid h; do
22      if( i != h ) then // do not inform itself about its position
23        h->GetPos()->SetE(i->GetPos()->GetI());
24      endif
25    done;
26  done;
27 done;
```

Coordinator updates all Synthetic Variables on Workers

In every iteration of the main loop, the Coordinator gets the up-to-date position of the ball (line 2). Then it queries all Workers for every Humanoid's position (lines 7 - 9). In the loop (lines 15- 26) the Coordinator updates all the synthetic Variables. First it updates the position of the ball (line 16) and then the position of every Humanoid (lines 21 - 24). The 'if' statement, (line 22), is to avoid updating the position of the current Humanoid.

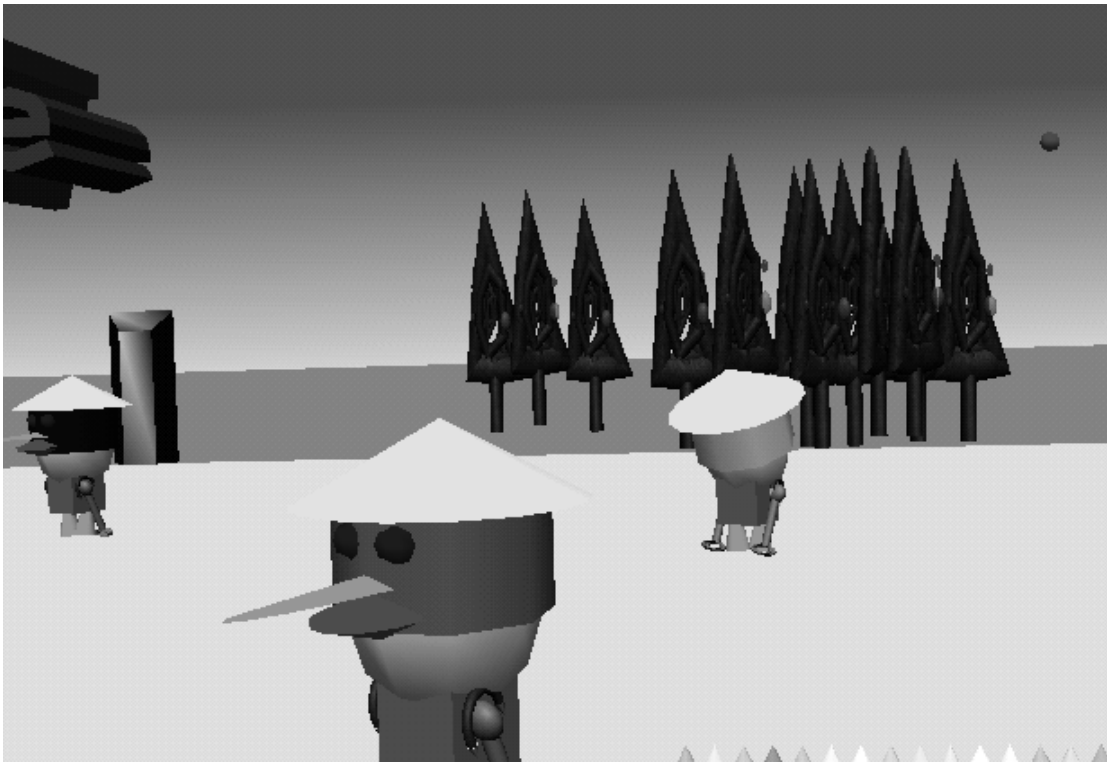
When multiple users participate in the virtual park, and thus multiple Coordinators, only the Master Coordinator updates the synthetic Variables. In this case the code in lines 15 - 26 is executed by the Master Coordinator only. This avoids the problem of updating the same Variables from multiple machines that could result in inconsistent values of the synthetic Variables.

A snapshot of the Virtual Park simulating 8 Humanoids (only seven are shown) in a non-distributed environment is shown below:



The DLoVe Virtual Park

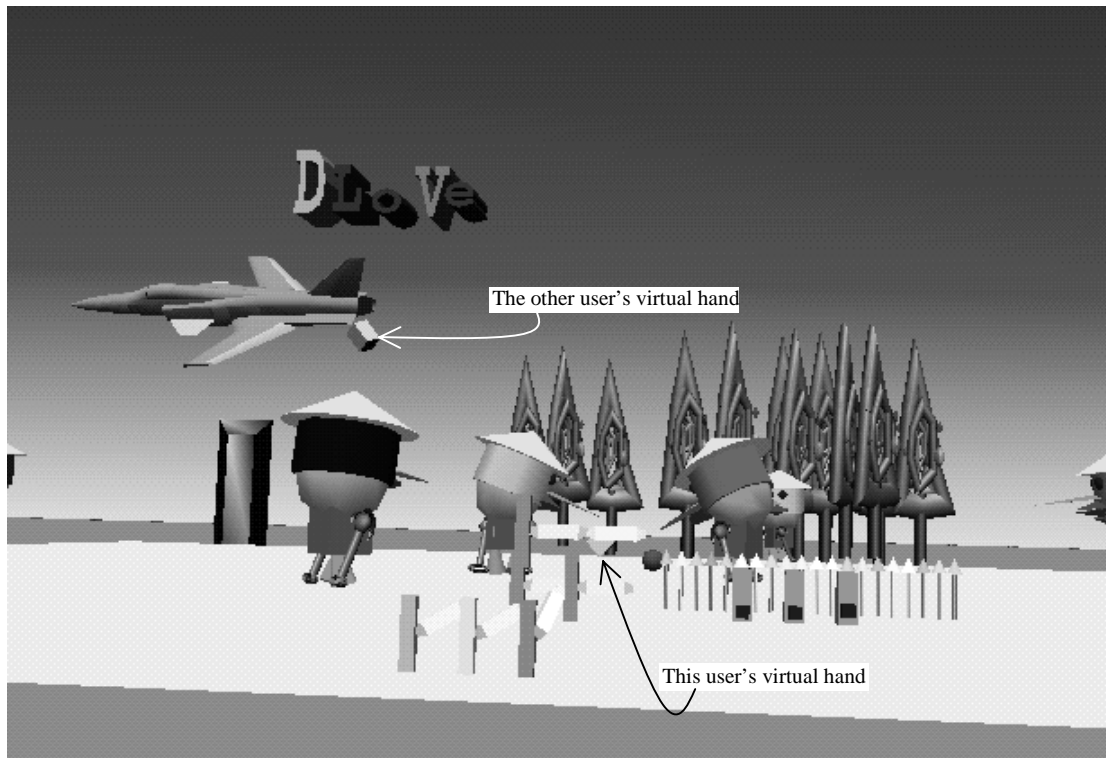
A snapshot of the same application in a two-user environment is shown below. One user is using an HMD and the Polhemus to interact, and the second user is using the mouse to manipulate the camera (position and orientation of the head) and also the virtual hand.



Master Coordinator using an HMD and the Polhemus to interact

Three Humanoids are visible here and the trees in the background. The Humanoid on the right has just hit the ball and is looking at it as it is traveling over the trees.

While this user is looking into the virtual park from this angle, the second user is trying to manipulate the controls. The following snapshot was taken at the same time I took the above one.



Slave Coordinator using the mouse to interact

In the above snapshot all the controls are visible at the front. The three sliders on the right that both users can manipulate to change the speed of the three orbiting objects are visible. In this snapshot only one of the orbiting objects is visible (the airplane top left). There are also three *ArmSlave* arms similar to the arms in the 'arms' program that follow the direction of the *Arm2* arm. The *Arm2* arm is the same as described in the 'arms' program in chapter 10 where one user can grab the first sub-arm and the other the second. And by pressing the 's' character the role is reversed and the first user can grab the second sub-arm and the second user the first.

Summary

The Virtual Park is a complete virtual world with a non-trivial level of complexity. The code examples presented here are the entire specification of the application for the desktop, the head-mounted display and the other variations on the application. Also, the modules needed to interface to external devices such as the Polhemus sensors and Performer rendering engine were presented.

It took me about 100 hours to develop the non-distributed version of the application where 30 hours were only for the 3 dimensional objects using external applications such as Showcase and i3dm. It then took me another 60 hours to develop the distributed and the multi-user versions of the application.

The goal of this programming exercise was to demonstrate that the DLoVe's claim, explored and supported by small test cases in chapters 10 and 11 continued to be valid when faced with a large scale, non-trivial, VR applications. According to the results of this exercise I think that this goal has been achieved.

Chapter 13: Performance Analysis

Overview

Two factors greatly influence the goodness of a graph partition. A given graph should be partitioned into concurrent modules to obtain the shortest possible program execution time. Secondly, one must choose the best size for each concurrent module that will result in fastest execution, while using a minimal number of processors. The grain-packing problem is related to optimal scheduling where an “optimal” scheduler executes in minimum time, well known to be NP complete in general.

DLoVe’s partition algorithm may be compared against load balancing and scheduling, but in reality it is someplace in between. Load Balancing, and more precisely Dynamic Load Balancing, tries to keep all processors equally busy, but does not try to reduce overall execution time. Only when the calculations are more expensive than the communication, the applications may run faster [Hesham 94].

Dynamic Load Balancing handles process migration and reacts to conditions that vary in the network. DLoVe's partition algorithm determines at compile time how to partition the constraint graph. DLoVe does not modify the partition of a given graph dynamically to efficiently implement Dynamic Load Balancing. It tries to partition the graph assuming that all Links are equally computationally expensive.

DLoVe assigns tasks to Workers in a manner similar to a scheduler. A single task in a scheduler corresponds to a single Link where in DLoVe a task corresponds of a set of interconnected Links as described in chapter 7. DLoVe uses the constraint graph itself to give extra information, making determining task allocation easier. Though a real scheduler allows tasks to have any cost, DLoVe assumes that all Links comprising a task have the same cost. This may cause DLoVe to create unbalanced task schedules but seems to work fine for many constraint graphs used in Virtual Reality.

DLoVe uses a greedy round-robin scheduling algorithm, which is not optimal in general. For example, assume there are 5 tasks with costs 3, 2, 2, 1, 1, and suppose there are two Workers available. Using greedy round-robin assignment, Worker 1 will be assigned tasks with costs $3+2+1=6$ while Worker 2 will be assigned tasks with costs $2+1=3$. An optimal scheduler might assign tasks of cost $3+1+1=5$ to Worker 1 and tasks of cost $2+2=4$ to Worker 2, with imbalance of 1 time unit.

Strategy for analysis

Two sets of experiments were conducted to measure the performance of DLoVe. The first set tested the performance of the Virtual Park while simulating 32 Humanoids. I measured the results of the non-distributed and the distributed versions using up to three Workers. To assess performance, I counted the *number of evaluations* of the 'SimulateLink' Link of each Humanoid routine for each Humanoid, as described in chapter 12. I also measured the resulting frame rate at which the Coordinator was able to render the graphics on the display. I purposely implemented a computationally expensive collision prevention algorithm for the Humanoids in order to overcome the network bandwidth bottleneck. This experiment employed a traditional approach for measuring performance: the overall computational throughput for the system.

The second set of experiments uses a new approach for measuring the performance of VR systems, by quantifying the accuracy of each frame rendered on the display. I defined the *accuracy* of a frame as the difference between the wall clock time and the minimum time since all output Variables (used for rendering the display) were last updated. Unlike the previous experiment, this one was conducted within a homogeneous environment. For this set of experiments I used a very simple simulation executed on Sun ultra 5 workstations running Solaris 2.7. I measured the latency of each network message as well as the number of requests initiated, the number of replies, and the number of messages that were discarded. This allowed me to compute the accuracy of each frame rendered. Results of this experiment show that traditional techniques for performance analysis do not accurately describe VR performance.

Performance with the Humanoids

As described in chapter 12, 'SimulateLink' simulates a Humanoid by advancing its position and running the collision prevention algorithm making a Humanoid look at the ball and wander in the park. Each 'SimulateLink' Link is executed at most once in every loop. To measure performance, I counted the total number of evaluations of this type of Link. Then I compared the results of the non-distributed version against the distributed version, utilizing between one and three Workers in the latter. For the distributed version only, I measured results for different values of the 'drop' threshold variable (discussed in chapter 7) to tune the 'Network Optimizer' module (discussed in chapter 8) of the Coordinator. I also compared frame rates with which the Coordinator was rendering the display in each case.

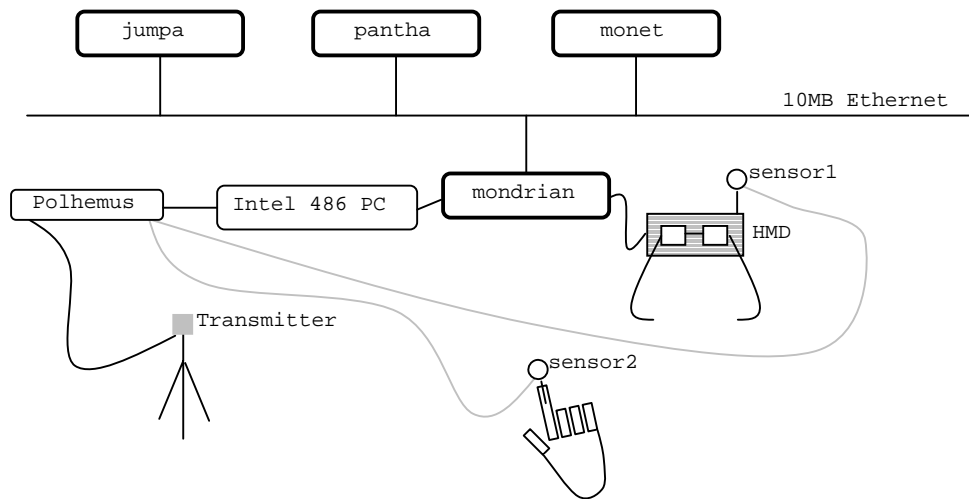
Collision Prevention Internals

The most computationally expensive Link in the Virtual Park application is the 'SimulateLink' Link. This Link takes care of the actions of a Humanoid as well as collision prevention. A Humanoid first tests to see if it is positionally close to any other Humanoids and then acts depending upon its identity and the identities of the other Humanoids close to it (as discussed in chapter 12).

Computer Network used for the Experiment

This experiment was conducted on Silicon Graphics workstations using SGI Performer as the graphics-rendering library. The four machines used had differing

hardware architectures. Thus the scheduler's task assumption that all Workers possess the same throughput is false, so that scheduling is less optimal than it would be in a homogeneous environment. The four machines, as well as the Polhemus (with the sensors and the transmitter) and the Eye tracking PC were connected as follows:



The SGI Experiment

The machine used for the graphics, for the distributed and non-distributed version, was 'mondrian'. All the others were used as Workers when running the simulation in the distributed environment. The PC above in the figure is the eye tracking hardware. For this experiment, even though I did not use the eye tracking hardware, I used the PC to read the Polhemus input though it.

The characteristics of the four machines are shown below, where the processor type, memory size, and cache differ between all four machines.

mondrian

1 250 MHZ IP22 Processor
FPU: MIPS R4000 Floating Point Coprocessor Revision: 0.0
CPU: MIPS R4400 Processor Chip Revision: 6.0
Data cache size: 16 Kbytes
Instruction cache size: 16 Kbytes
Secondary unified instruction/data cache size: 2 Mbytes on Processor 0
Main memory size: 64 Mbytes
Graphics board: High Impact

jumpa

1 195 MHZ IP28 Processor
CPU: MIPS R10000 Processor Chip Revision: 2.5
FPU: MIPS R10010 Floating Point Chip Revision: 0.0
Data cache size: 32 Kbytes
Instruction cache size: 32 Kbytes
Secondary unified instruction/data cache size: 1 Mbyte
Main memory size: 128 Mbytes
Graphics board: High Impact

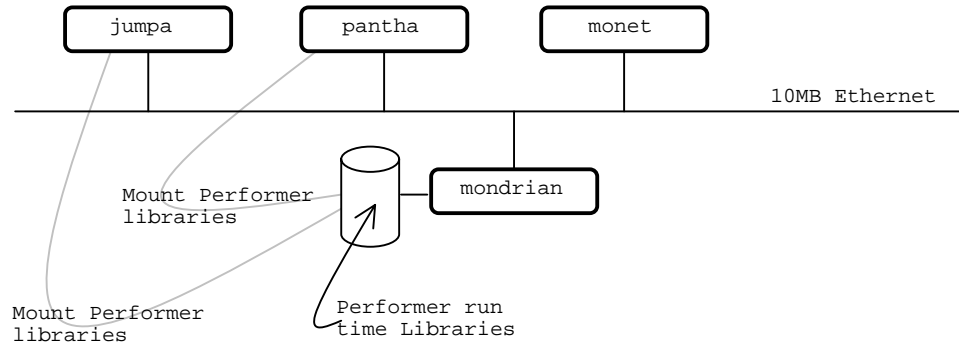
pantha

1 250 MHZ IP22 Processor
FPU: MIPS R4000 Floating Point Coprocessor Revision: 0.0
CPU: MIPS R4400 Processor Chip Revision: 6.0
Data cache size: 16 Kbytes
Instruction cache size: 16 Kbytes
Secondary unified instruction/data cache size: 2 Mbytes on Processor 0
Main memory size: 128 Mbytes
Graphics board: High Impact

monet

1 200 MHZ IP22 Processor
FPU: MIPS R4000 Floating Point Coprocessor Revision: 0.0
CPU: MIPS R4400 Processor Chip Revision: 6.0
Data cache size: 16 Kbytes
Instruction cache size: 16 Kbytes
Secondary unified instruction/data cache size: 1 Mbyte on Processor 0
Main memory size: 64 Mbytes
Graphics board: GUI-Extreme

'Jumpa' and 'Pantha' mounted the Performer run time libraries of 'mondrian', via the Network File System (NFS), adding even more overhead to the network and to 'mondrian', 'monet' has its own copy of the libraries:



Mounting Performer run time libraries

Analyzing the Results

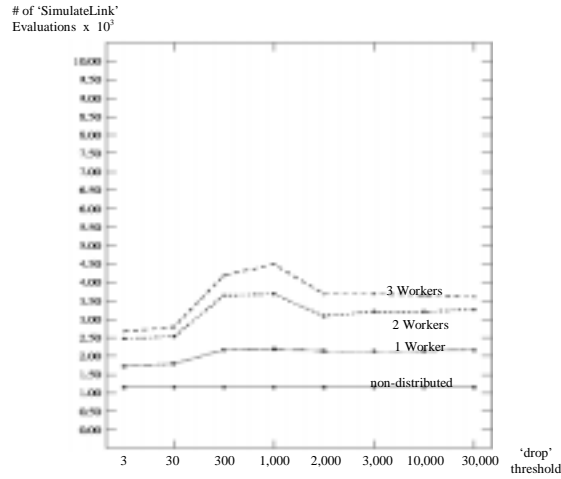
Two variables were varied during measurement of distributed version: the number of Workers participating, and the 'drop' threshold variable. As it was discussed in chapter 9, the Coordinator needs to disregard messages that it builds when there are many pending requests on the network. As a result, for this experiment I first ran the non-distributed version, counting the number of the 'simulateLink' Link evaluations and measuring the frame rate. Then I ran the distributed version with one, two, and three Workers. Finally I repeated the distributed version experiment with a new value for the 'drop' threshold.

Recall that the 'drop' threshold is used to limit the number of requests sent from the Coordinator to Workers and thus avoid overloading the network. If this threshold is too small, then Workers will not receive enough tasks to keep them busy. If the threshold is too large then two conditions may arise. The Coordinator may overload the network with Worker requests, and eventually crash, or it may spend most of its time trying to send requests to Workers, thus spending less time rendering the display, reading replies from the Workers, and reading input devices. On the other hand, the Workers may be so busy serving the Coordinator's requests that all replies come back to the Coordinator too late to be useful. In Virtual Reality, when most of the replies come back to the Coordinator too late there is no real-time interaction and the result cannot be called Virtual Reality anymore, even if the frame rate stays high. In this case frame rate alone does not accurately describe the efficiency of a rendering system.

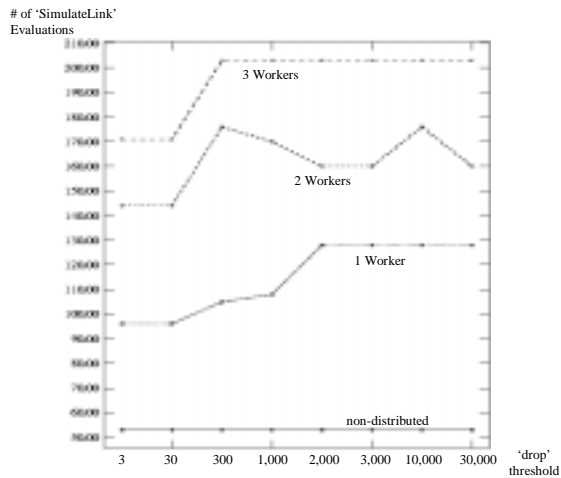
For this set of experiments I utilized one, two, and three Workers. 'Jumpa' was always the first Worker selected, then 'pantha', and finally 'monet' in this order. For example, when two Workers were used for the simulation, 'jumpa' was the first Worker to connect to the Coordinator and 'pantha' was the second one. This way the first, third, fifth, etc Humanoids were simulated by 'jumpa' and the second, fourth, sixth, etc Humanoids were simulated by 'pantha'.

Since there are 32 Humanoids, when 3 Workers are utilized for the simulation 'jumpa' and 'pantha' simulate 11 Humanoids where 'monet' simulates only 10. Recall that the most powerful machine is 'jumpa' and the least powerful one is 'monet'. If the same experiment is conducted starting 'monet' first or second the results might be different.

The following graphs show the results of running the application in non-distributed mode and in distributed mode using one, two, and three Workers. The horizontal axis shows the value of the 'drop' threshold, while the vertical axis shows the number of evaluations of the Link 'SimulateLink'. One must note the difference in the Y-axis scale when reading the following two graphs:



Comparing the number of evaluations of the non-distributed and the distributed version of the Virtual Park application using computational expensive evaluations. 'SimulateLink' is $O(n^3)$ where n is the number of Humanoids (32).



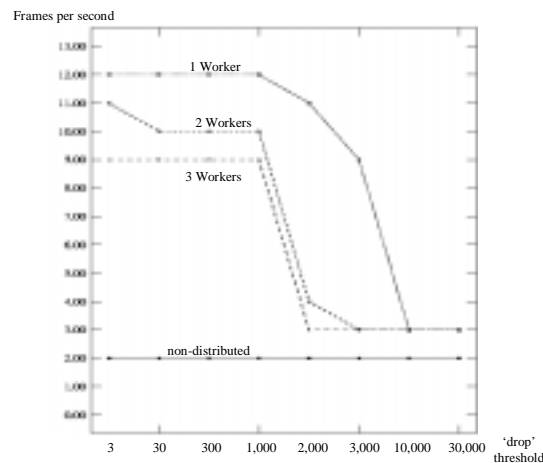
'SimulateLink' is $O(n^4)$ where n is the number of Humanoids (32).

In this experiment, by increasing the number of Workers, we increase the number of evaluations and thus seem to observe better performance. This is because in the

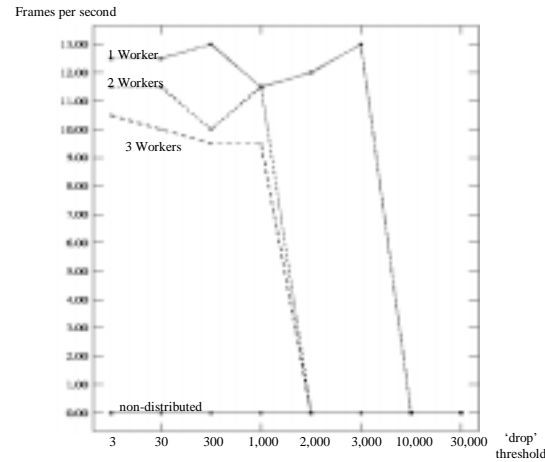
non-distributed version, a single machine is trying to simulate all 32 Humanoids, where in the distributed version the workload is given to multiple machines simulating the Humanoids “in parallel”. The distributed version with one Worker outperforms the non-distributed version because the Worker simulates the Humanoids while the Coordinator handles all user’s requests and the drawing of the scene. Also in the distributed version I always used the most powerful Worker for the most complex tasks, then the next powerful one, and so on.

We can also see the effect of the ‘drop’ threshold variable. To tune this threshold, the designer needs to run several simulations to figure out the best value for the specific application and the specific network. From this experiment we can see that the most evaluations are taking place when we use three Workers and we set the ‘drop’ threshold around 1000.

However, the more Workers we utilize, the fewer frames per second we get:



Number of frames when the running time of ‘SimulateLink’ is $O(n^3)$, where n is the number of Humanoids



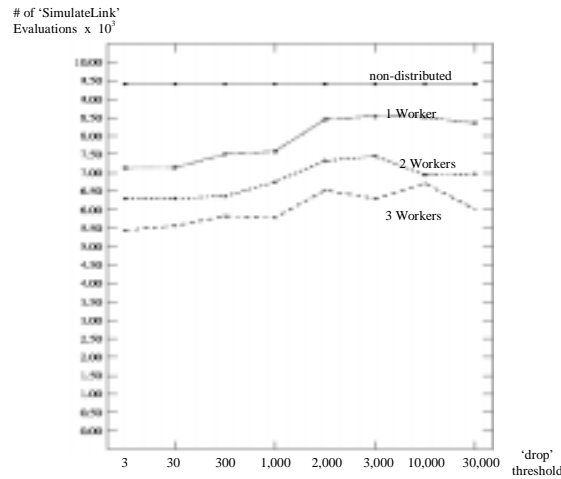
Number of frames when the running time of 'SimulateLink' is $O(n^4)$, where n is the number of Humanoids

When we use more Workers, we get fewer frames per second because the Coordinator saturates the network faster due to sending the same messages to multiple Workers. The Coordinator sends the same messages to all Workers because DLoVe is build on top of TCP/IP simulating multicasting. For example, when there are three Workers the Coordinator has to send the same message three times, once for each Worker. This adds a lot of overhead to the Coordinator and as a result it spends less time updating the display. The greatest decrease in frame rate is when the 'drop' threshold is over 2000. This is because the Coordinator is trying to write too many messages on the network, and the network becomes saturated and network resources become temporarily unavailable. The Coordinator keeps trying until it successfully writes the messages on the network, losing critical time on updating the display. Several times the application crashed when I was using one Worker with a 'drop' threshold of over 2000.

The distributed version outperformed the non-distributed version in frame rate, because in the non-distributed version one machine is trying to simulate all 32 Humanoids, leaving it no time to update the display. In fact, the non-distributed version was so slow (2 frames per second) that it would disorient any person using it. The frame rate on the distributed version could be improved if a different technique was used, for network communication, such as multicasting instead of unicasting. If DLoVe outperforms the non-distributed version using TCP/IP for point-to-point communication, it promises even greater performance if it is re-implemented using multicasting.

The above experiment demonstrates that by using more Workers we get more evaluations, but fewer frames per second. Because the 'SimulateLink' evaluation is computationally very expensive (more precisely the 'CalculateForce()' function described in chapter 12) the application computation load is the critical bottleneck rather than the network. This means that the time needed to simulate the Humanoids is much greater than the cost of sending the requests over the network.

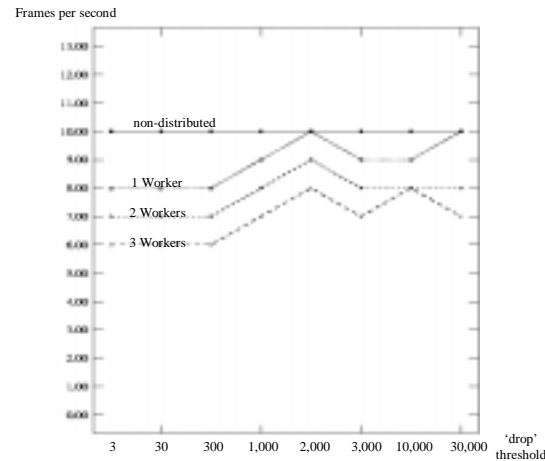
To study what would happen if 'SimulateLink' was inexpensive, I re-ran the same experiment without the 'CalculateForce()' function, allowing the Humanoids to collide with each other. In this case the non-distributed version of the application outperformed the distributed versions. Also the more Workers I used, the fewer evaluations I received:



Comparing number of evaluations of the non-distributed and the distributed version of the Virtual Park application using computational inexpensive – $O(n)$ – evaluations.

This is because sending a request over the network is more computationally expensive than doing the evaluation locally. Thus the more Workers I used, the more time the Coordinator lost in sending requests to each Worker.

In addition, the more Workers the Coordinator used, the fewer frames it rendered, because it was spending most of its time trying to send/receive requests on the network:



Comparing the frame rate of the non-distributed and the distributed version of the Virtual Park application using computational inexpensive – $O(n)$ – evaluations

Another issue I encountered in both experiments when the 'drop' threshold was somewhere over 3000 is that replies from the Workers exhibited large latencies. Even though the number of evaluations increases, as the 'drop' threshold increases, all replies became older. In Virtual Reality, this means that when the user moves his hand, the system responds after many milliseconds, giving the impression that the user's hand is not really attached to the virtual hand. Or the user may turn his head and after some considerable wait, the system draws the correct perspective of the view.

The actual latency of replies was not measured by this experiment, although it led us to suspect that latencies were abnormally large.

I tried to run the same experiment with two users but encountered two problems. The second machine I had available was not attached to a Polhemus tracker. This problem was resolved by simulating a second Polhemus with a mouse (this is

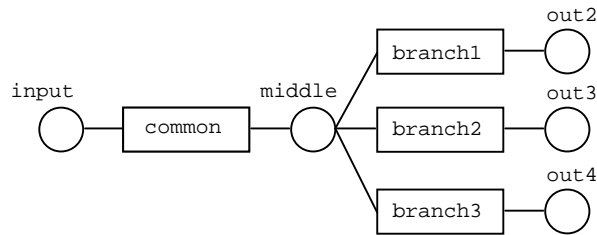
something that DLoVe supports in its framework). The second problem was that this second machine was a very slow Silicon Graphics workstation where the rendering was done in software; whereas 'mondrian' uses hardware for rendering. This second machine was so slow that even without using the 'CalculateForce()' function, I was only getting 40% of the evaluations of the same application using 'CalculateForce()' on 'mondrian'.

Performance with the 'Perf' program

The analysis of the Virtual Park, even though it illustrated some of the positive and negative aspects of DLoVe was conducted within a heterogeneous environment. The outcome of the experiment might have been different if all machines were exactly of the same architecture and all possessed a local copy of the Performer run-time libraries. Clearly I needed a more carefully conducted procedure for conducting the experiment to get more conclusive and more accurate results.

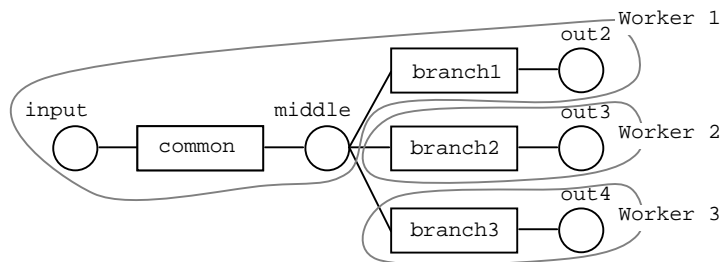
To address this, I implemented a very simple "Perf" application that was executed in a homogeneous environment, and made accurate measurements of its performance. I was able to measure latency of individual messages sent by the Coordinator, and the number of messages that were built and discarded by the Coordinator due to the setting of the 'drop' threshold.

The 'Perf' benchmark program consists of 4 Links and 5 Variables. It was designed so that DLoVe can partition it into at most 3 Workers. The following figure shows the Links and Variables within this application:



DLove graph of the 'perf' application

The 'main' loop of the application first sets the Variable 'input' and then requests the values of the three output Variables (out2, out3, and out4). When this program is executed using three Workers, each Worker is assigned one output Variable as shown below, and thus the Coordinator can request these three output Variables in parallel from the three Workers:



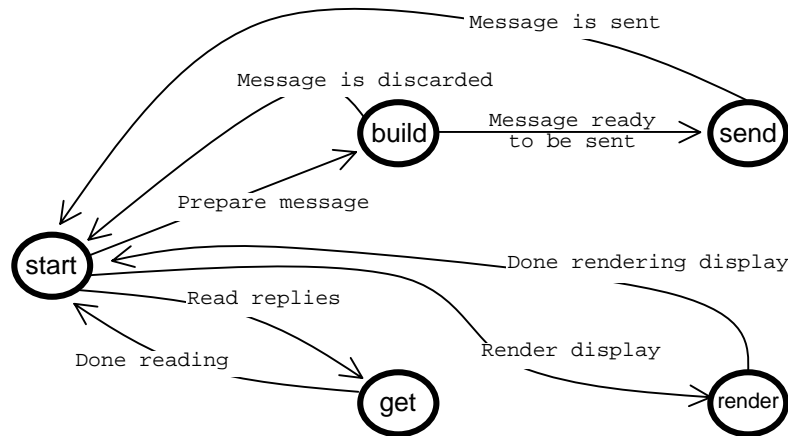
Partition of the DLoVe graph of the 'perf' application

State machine of the run time system

The Coordinator can be thought of as possessing five independent states. The Coordinator builds a message consisting of several requests and sends it out to the Worker(s). Initially, the Coordinator is in a 'start' state. When it sees a need to request set or get a Variable's value or to enable or disable a Link, it transitions to

the 'build' state within which it builds the message that it may send to the Workers. This message consists of requests such as SetE, GetE, Enable, and Disable. At the end of the main loop, just before sending the message, it checks its internal counters, which indicate the number of pending requests on the network to each Worker. If a counter is below a chosen threshold, it transitions to 'send' state, sends the message, and returns to the 'start' state. Else, it discards the message and returns to the 'start' state.

While in the 'start' state, it checks to see if any replies came back from the Workers. If replies have arrived, it transitions to the 'get' state and processes all the replies. When all replies are processed, it transitions back to the 'start' state where it starts building messages all over again. The 'start' state is a state indicating idle time and it is used as a starting point in describing the functionality of the Coordinator. The fifth state is the 'render' state in which the Coordinator transitions to render the display. The following figure shows the five states and the transitions between them:

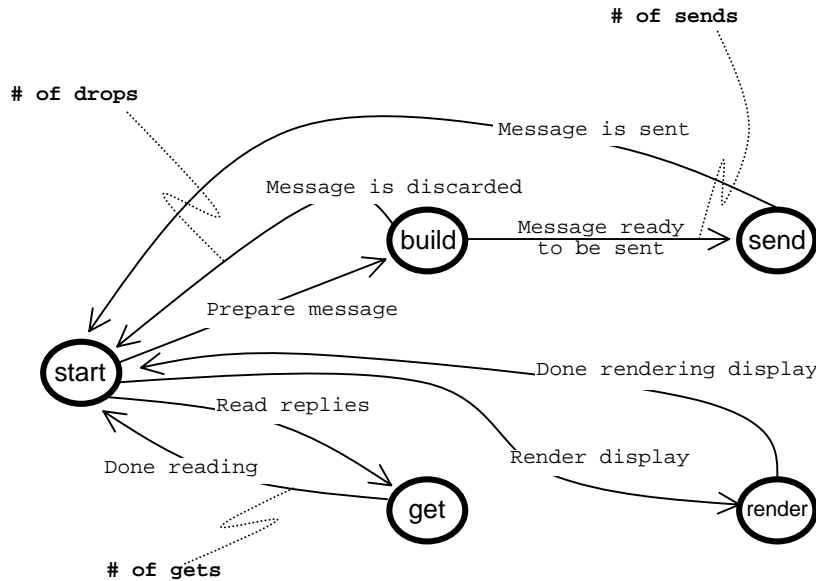


Internal state machine of DLoVe when executed in the distributed mode

What is Measured

The experiments were run for 60 seconds, where DLoVe collected two kinds of performance statistics every second. Statistics included the number of requests initiated by the Coordinator, the number of messages that were discarded, and the number of replies that came back to the Coordinator. Additional performance data included latency of each request. Every request was timestamped with the time of initiation before being sent to the Worker(s). The Worker that responded to the requests preserved the timestamp in its reply. The Workers also marked a GetE reply with the amount of time it took the Worker to update the requested Variable. When the Coordinator received the reply, it calculated the elapsed time of the request and subtracted the amount of time the Worker needed to update the requested Variable. I implemented utilities that utilize the `xgraph` program to plot the network latency for every request, as well as the computational time a Worker took to update the requested Variable.

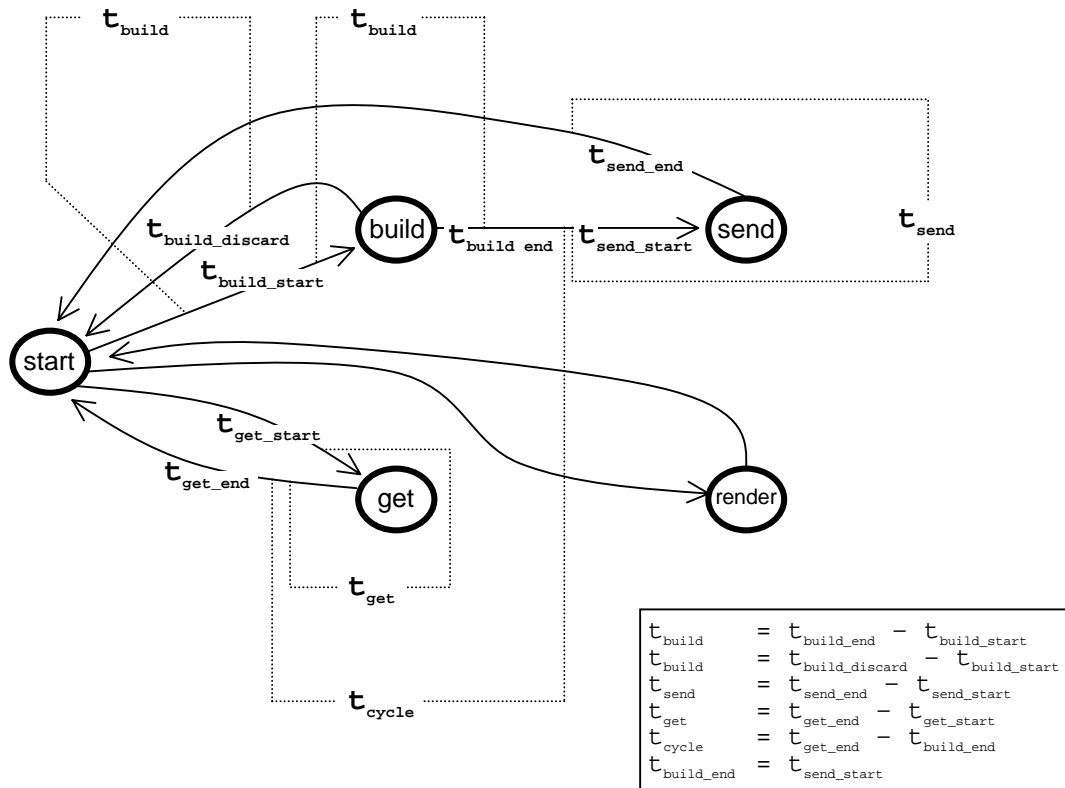
The following figure shows the state transitions where I measure how many messages the Coordinator sends, how many it drops, and how many it receives every second:



Measurement of DLoVe

Performance in time required measuring the elapsed time between pairs of state transitions as shown in the following figure. t_{build} is the time it takes the Coordinator to build a message before sending it to the Workers or discarding it, which is the difference between t_{build_end} and t_{build_start} , or between t_{build_start} and $t_{build_discard}$ representing the time at which the build ended and when it started. t_{send} is the time it takes the Coordinator to put the message on the network, which is the difference between t_{send_end} and t_{send_start} indicating the time the Coordinator completed the transmission of the message and the time it started transmitting. t_{get} is the time it takes the Coordinator to process the replies from the Workers, which is the

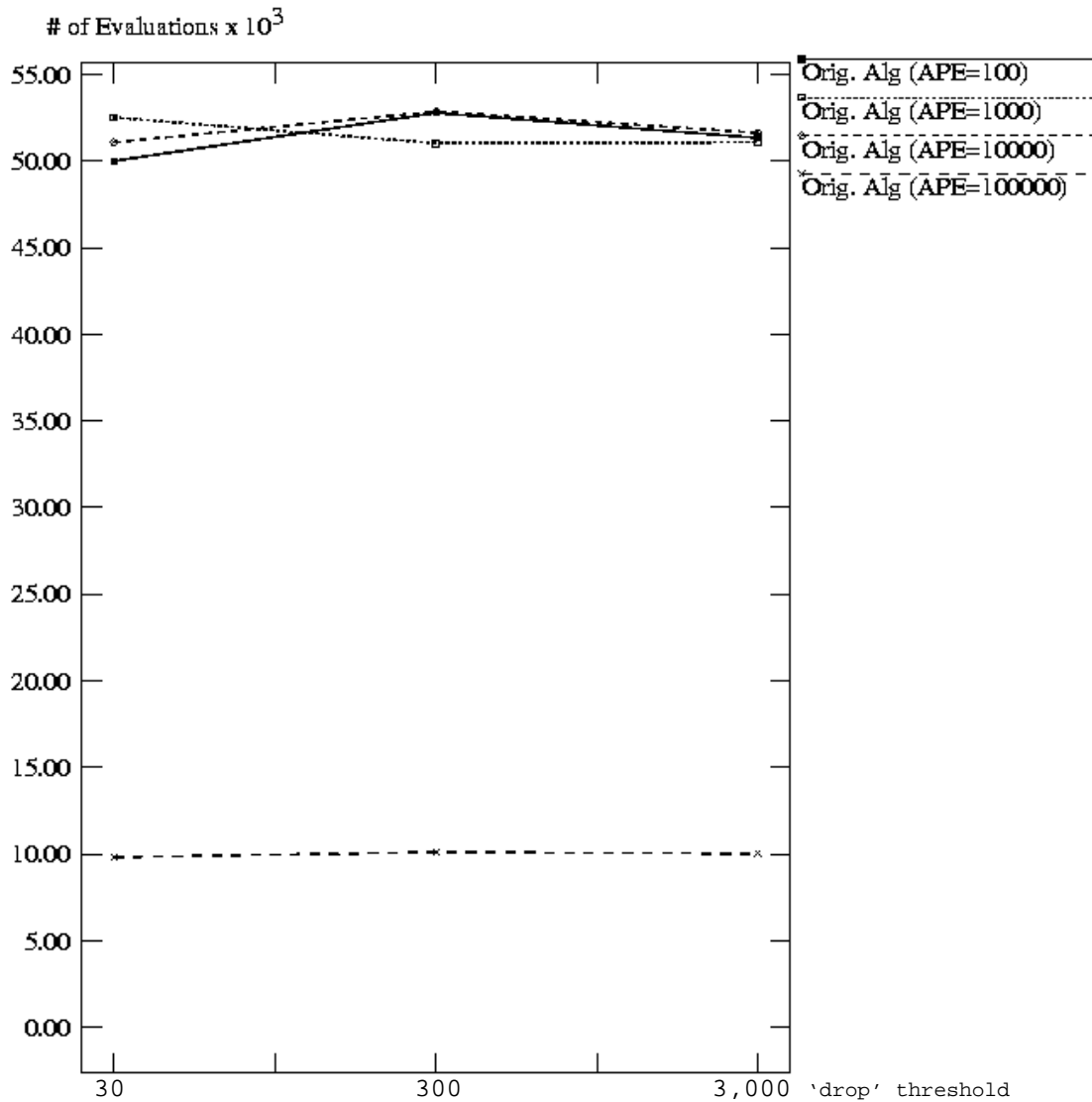
difference between t_{get_start} and t_{get_end} indicating the time it started reading replies and the time it read all replies from all Workers. t_{cycle} indicates the time it takes a request to come back as a reply, which is the difference between t_{build_end} and t_{get_end} . This is the latency of each message and this is what in which I am the most interested. That the Coordinator receives many replies from the Workers is a goal I would like to achieve. However, if the messages are all too old, then user interaction is minimized and this is not what I want since DLoVe is designed for real-time applications such as Virtual Reality.



Measuring latency in the internal state machine

Analyzing the Results

As in the previous experiment in the Virtual Park, the following graph shows that increases in Workers provide increases in throughput.

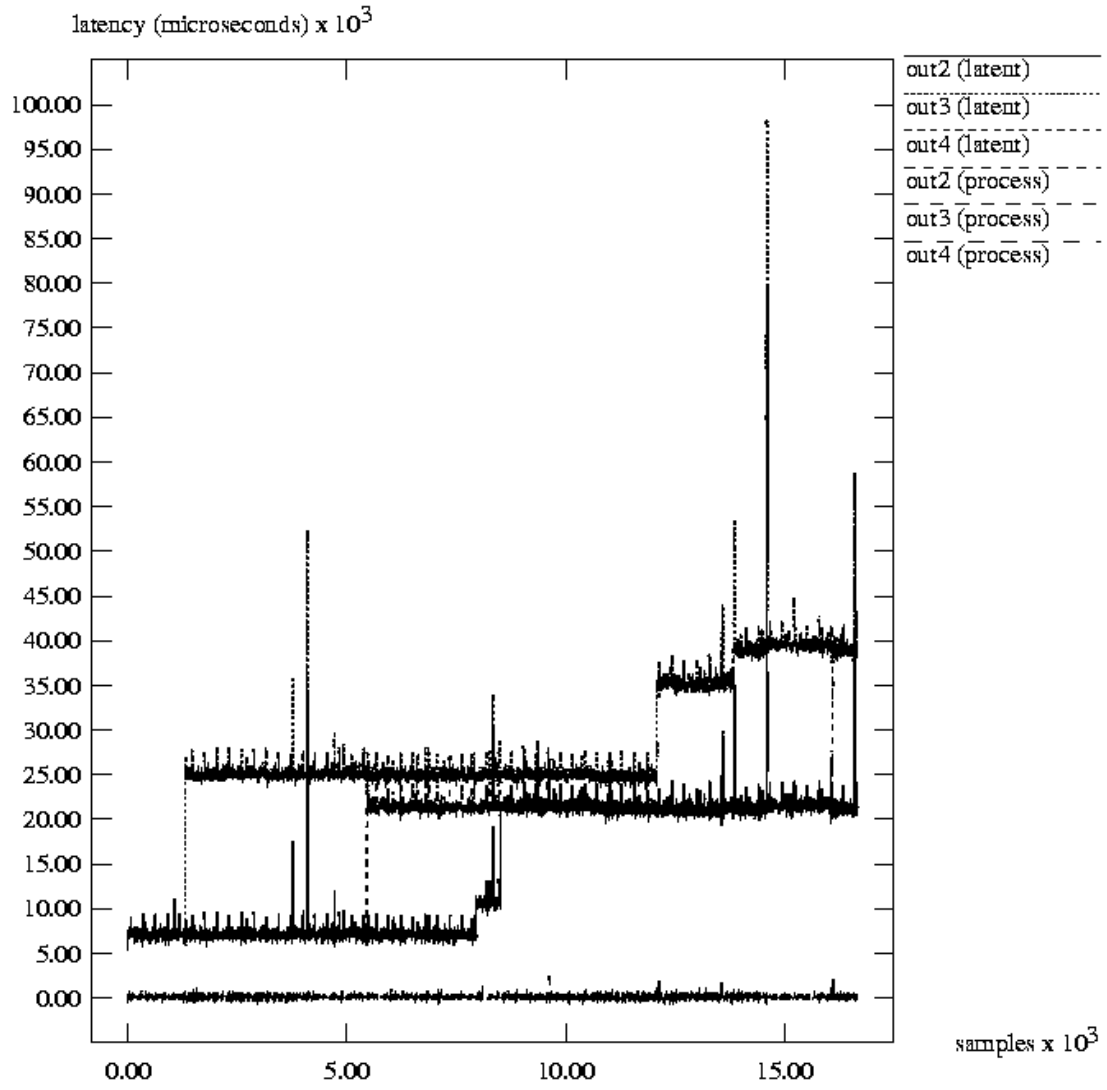


Throughput of DLoVe with different 'drop' and APE

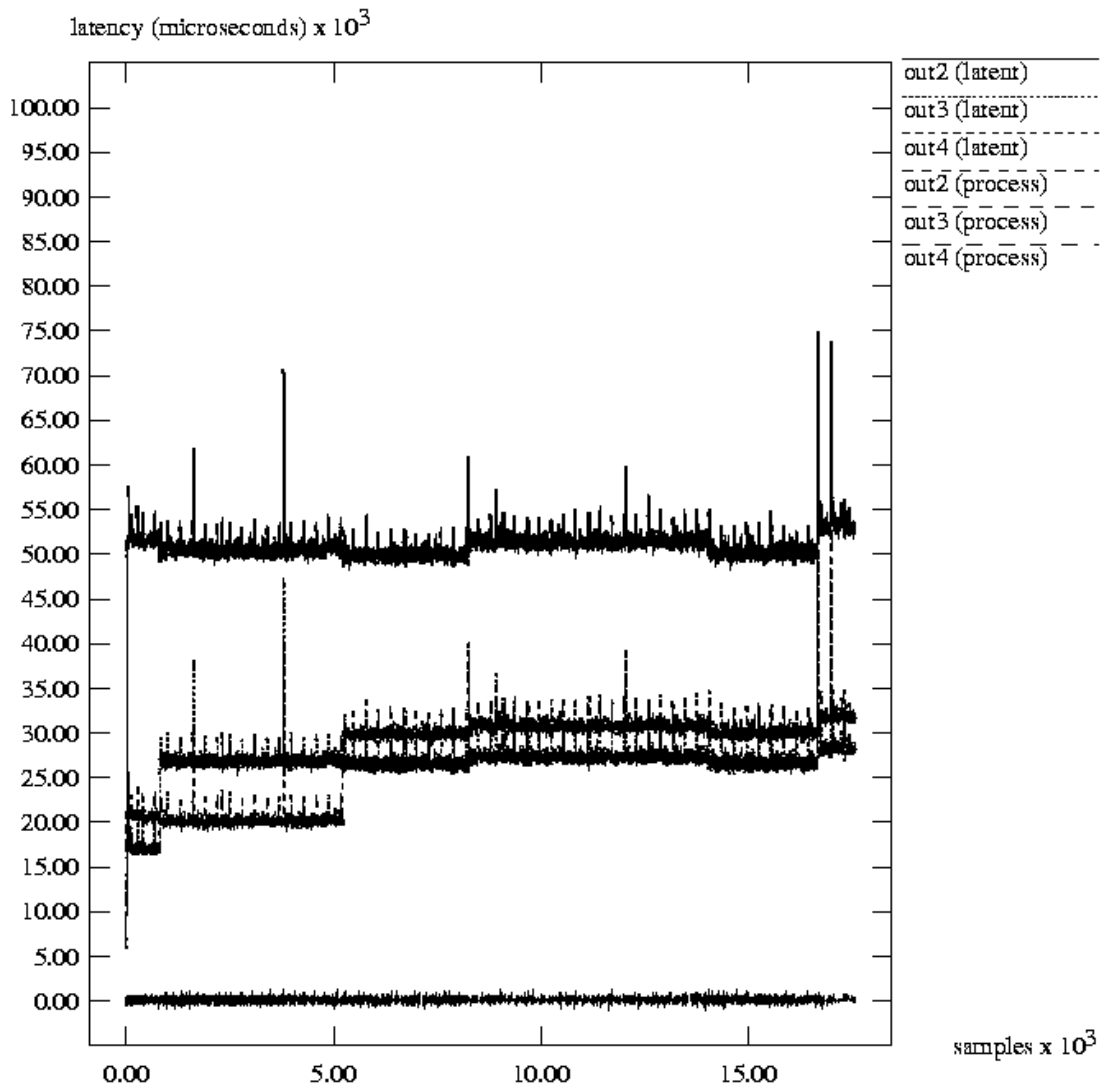
However, throughput of a Virtual Reality system does not describe how well it performs. A more critical factor in VR is how high a frame rate it can achieve, and how accurately each frame represents the virtual world.

Accuracy of rendition is difficult to measure. In DLoVe, the Coordinator issues GetE requests to update certain Variables to render the display. These requests are sent to the Workers that update the requested Variables, which then send the results back to the Coordinator. These replies are not instantaneous, but arrive with some latency that depends on performance upon the network. So, one measure of accuracy is message latency. Another is how up-to-date the frames are when rendered, relative to user input and the real state of the virtual world.

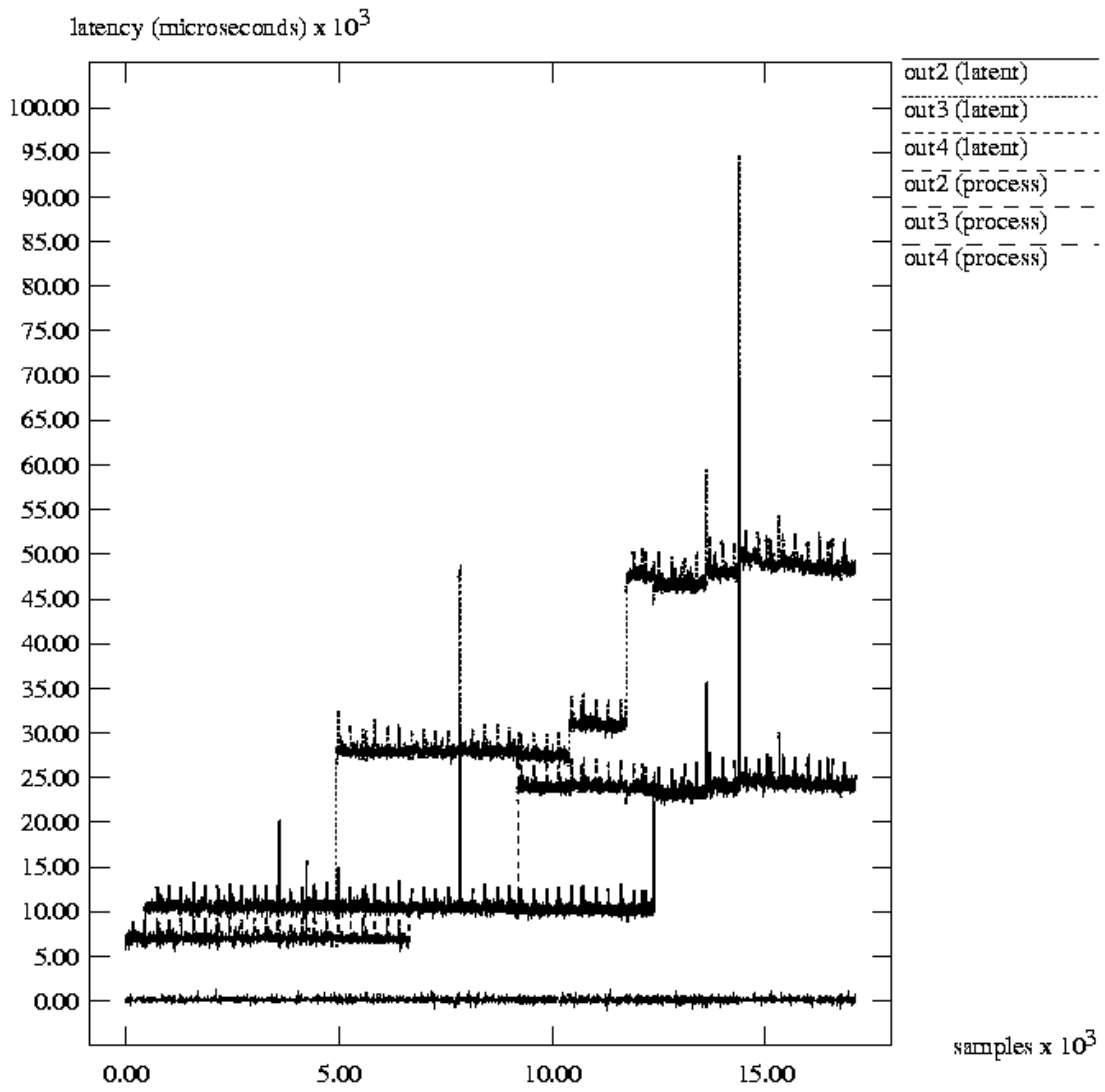
The following 12 graphs show the latency of each request sent to each Worker. Each request is time-stamped before it is sent. Workers also time-stamp each request with the amount of time taken to bring a requested Variable up-to-date. When the Coordinator receives the requested Variable from a Worker, it calculates the elapsed time of the request minus the amount of time a Worker needed to bring the requested Variable up-to-date. This yields to the amount of time a request spends on the network. For this experiment I utilized 3 Workers and tuned two variables: 'drop' threshold and expensiveness of the 'branch' Link. I simulated expensiveness of the 'branch' Links by performing multiple floating-point additions. I call this variable "Additions Per Evaluation" (APE). The APE of each 'branch' Link took the following values: 100, 1,000, 10,000, and 100,000. For each of these values, I used a 'drop' threshold of 30, 300, and 3,000, yielding a total of 12 experiments. These experimental measurements showed abrupt changes in average latency over time, which at first puzzled me. Note also the different scales for all these graphs.



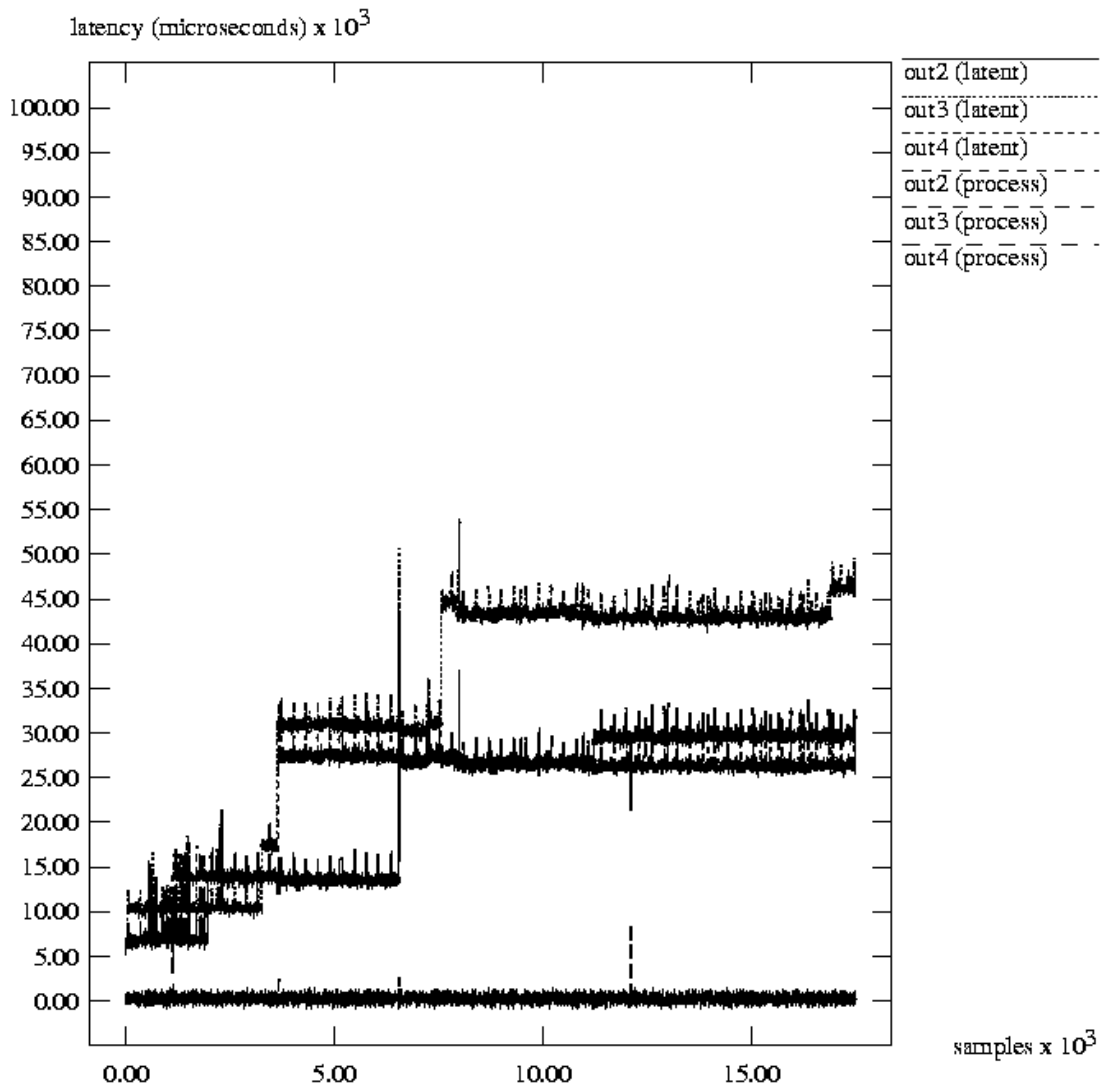
GraphID(001L) drop=30, APE=100



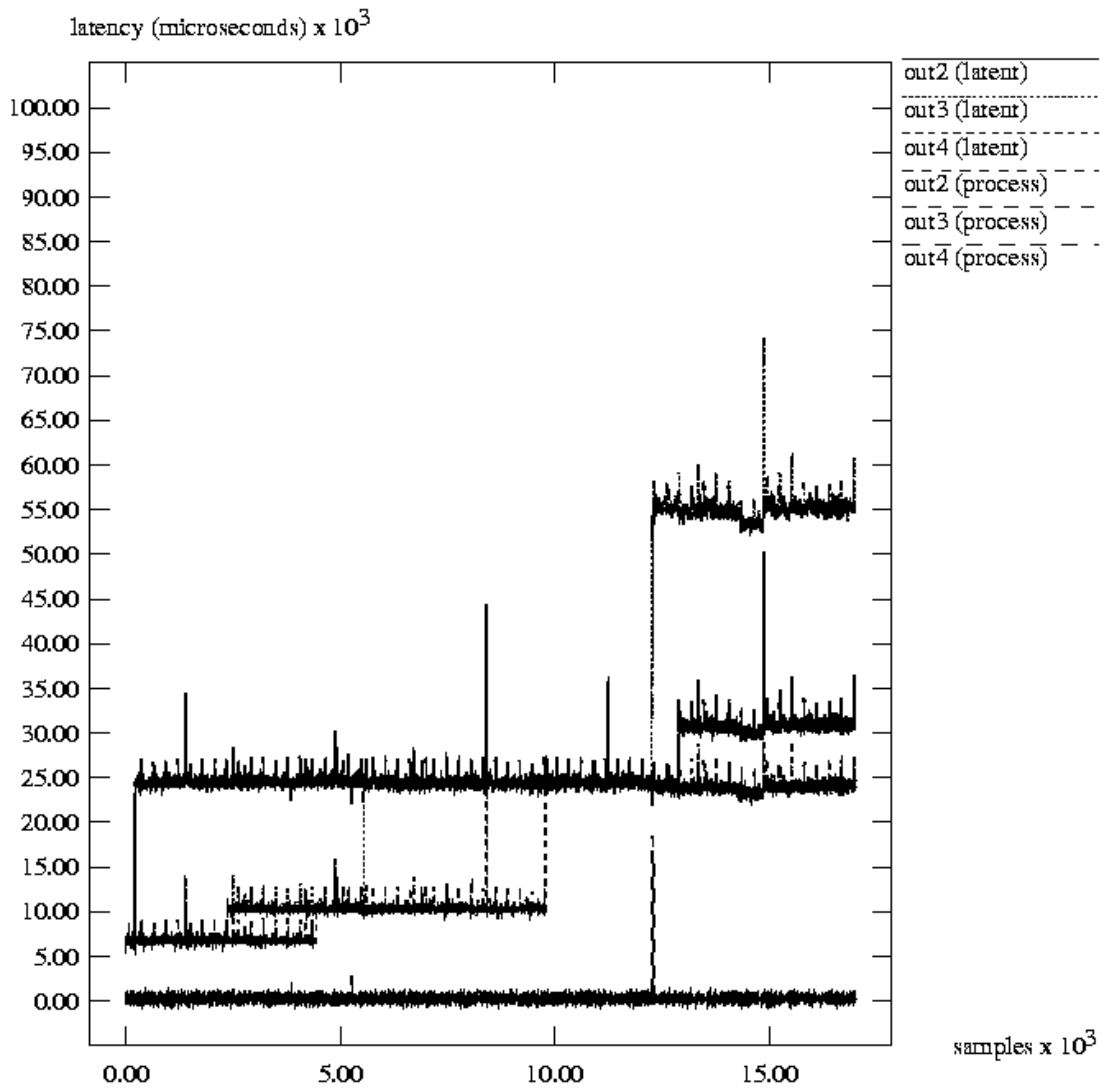
GraphID(002L) drop=300, APE=100



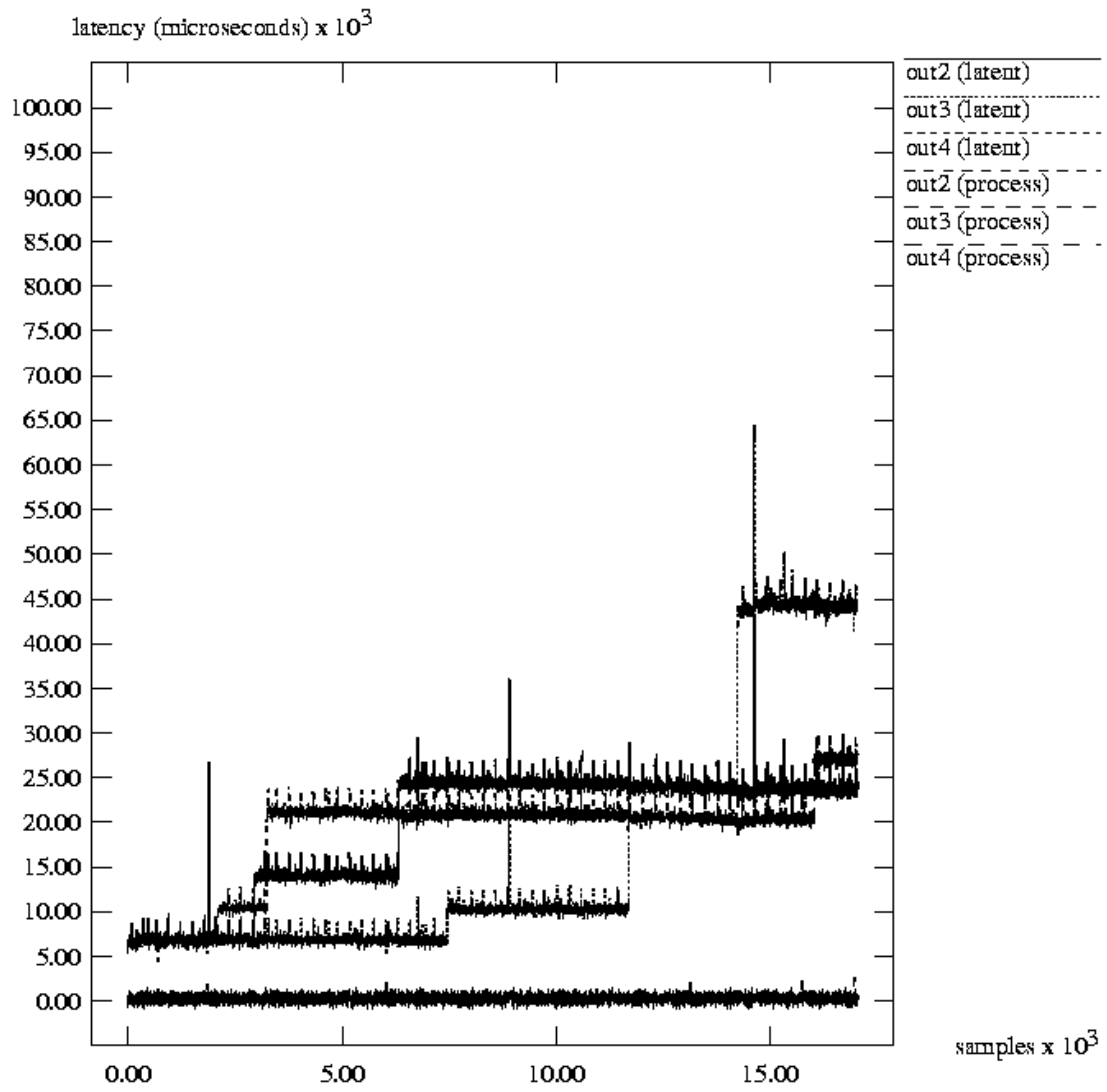
GraphID(003L) drop=3000, APE=100



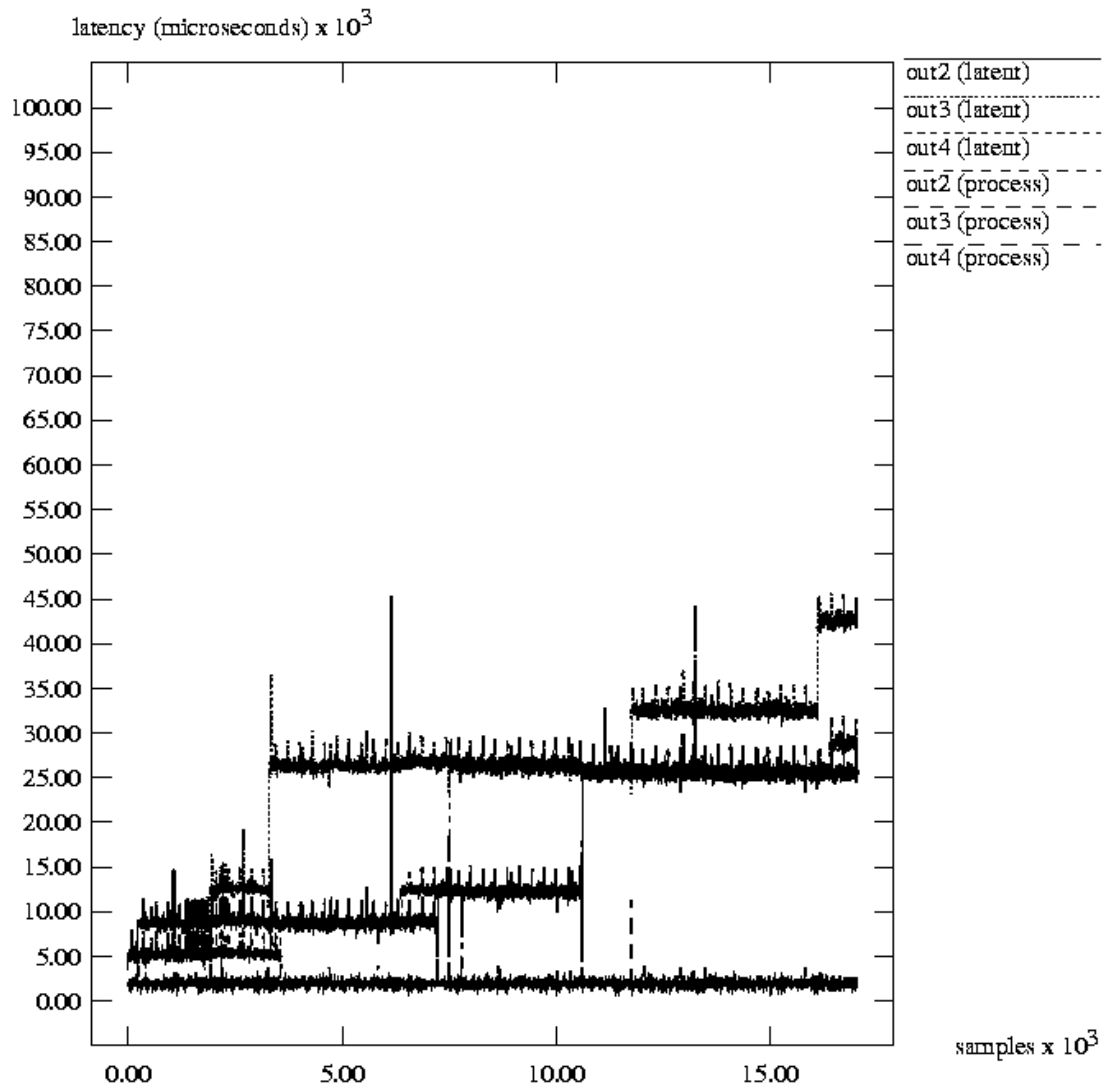
GraphID(004L) drop=30, APE=1000



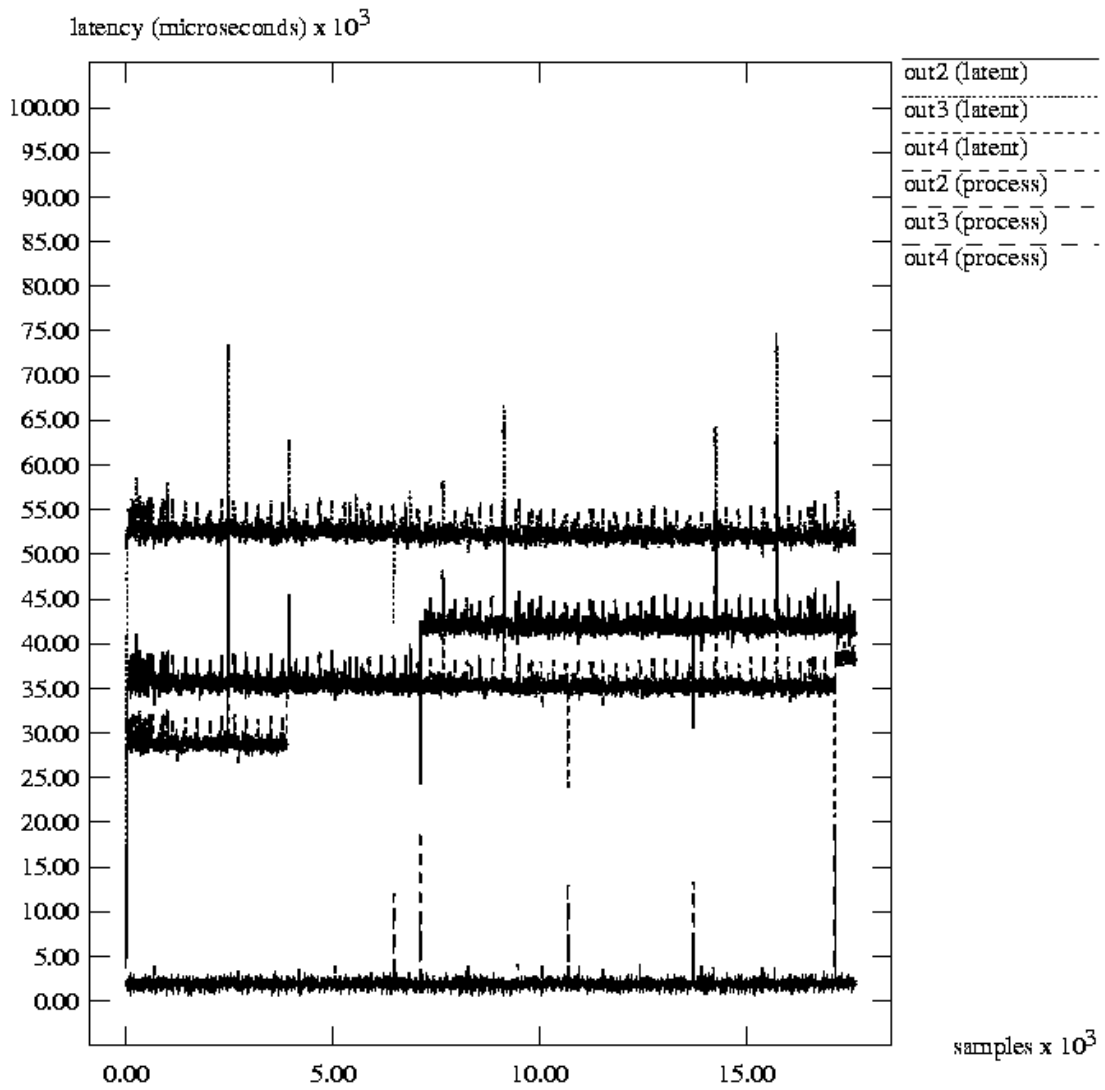
GraphID(005L) drop=300, APE=1000



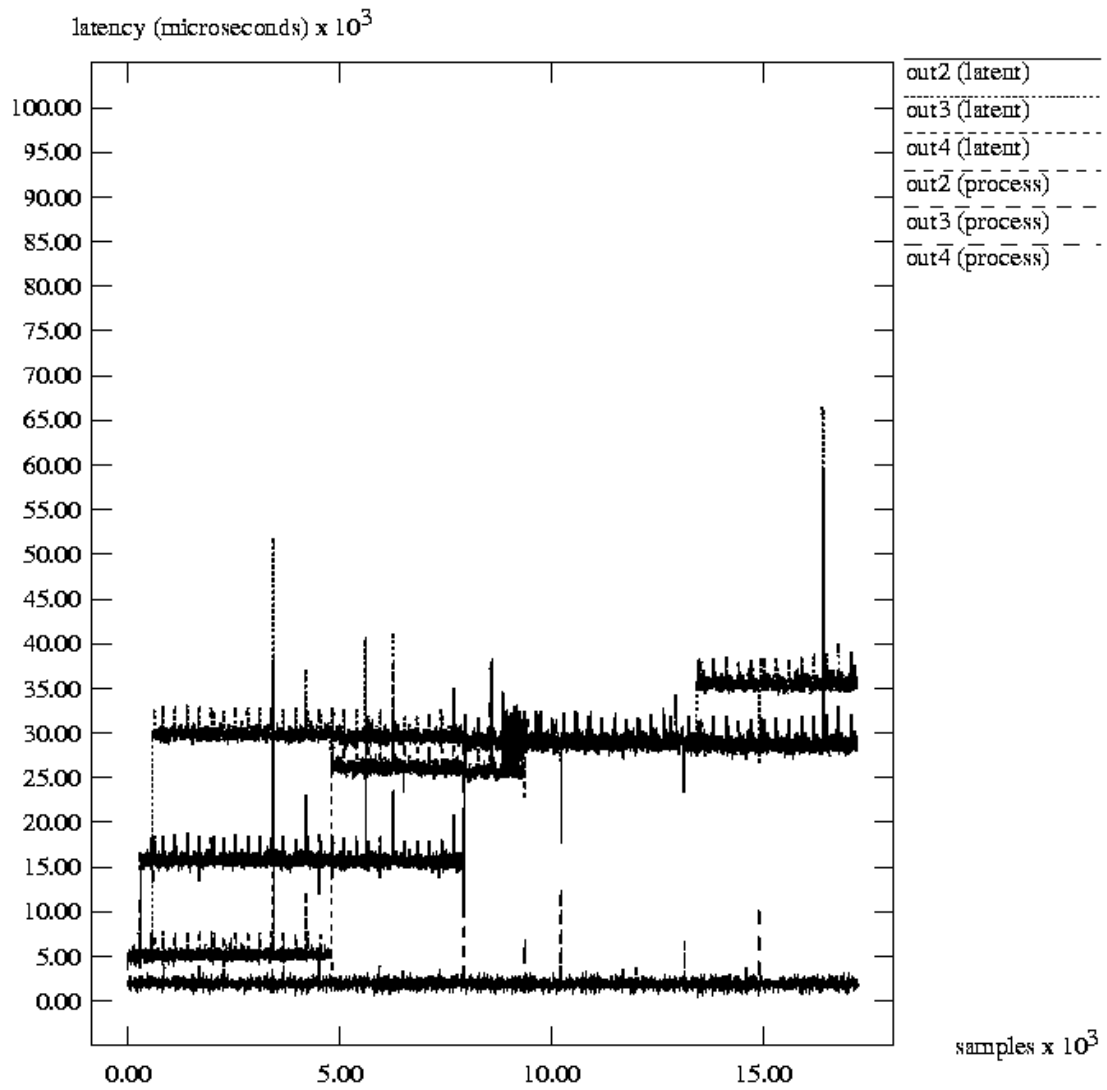
GraphID(006L) drop=3000, APE=1000



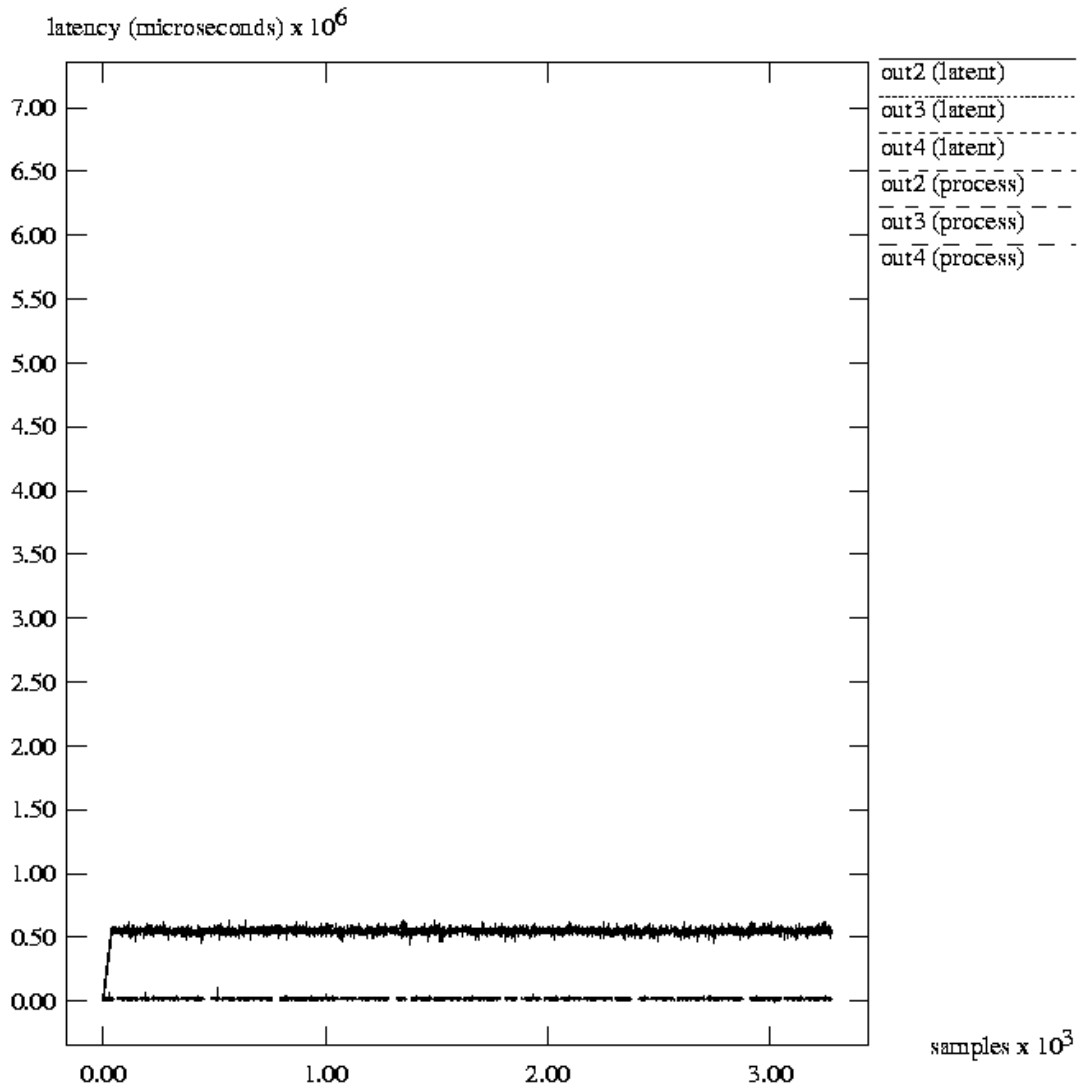
GraphID(007L) drop=30, APE=10000



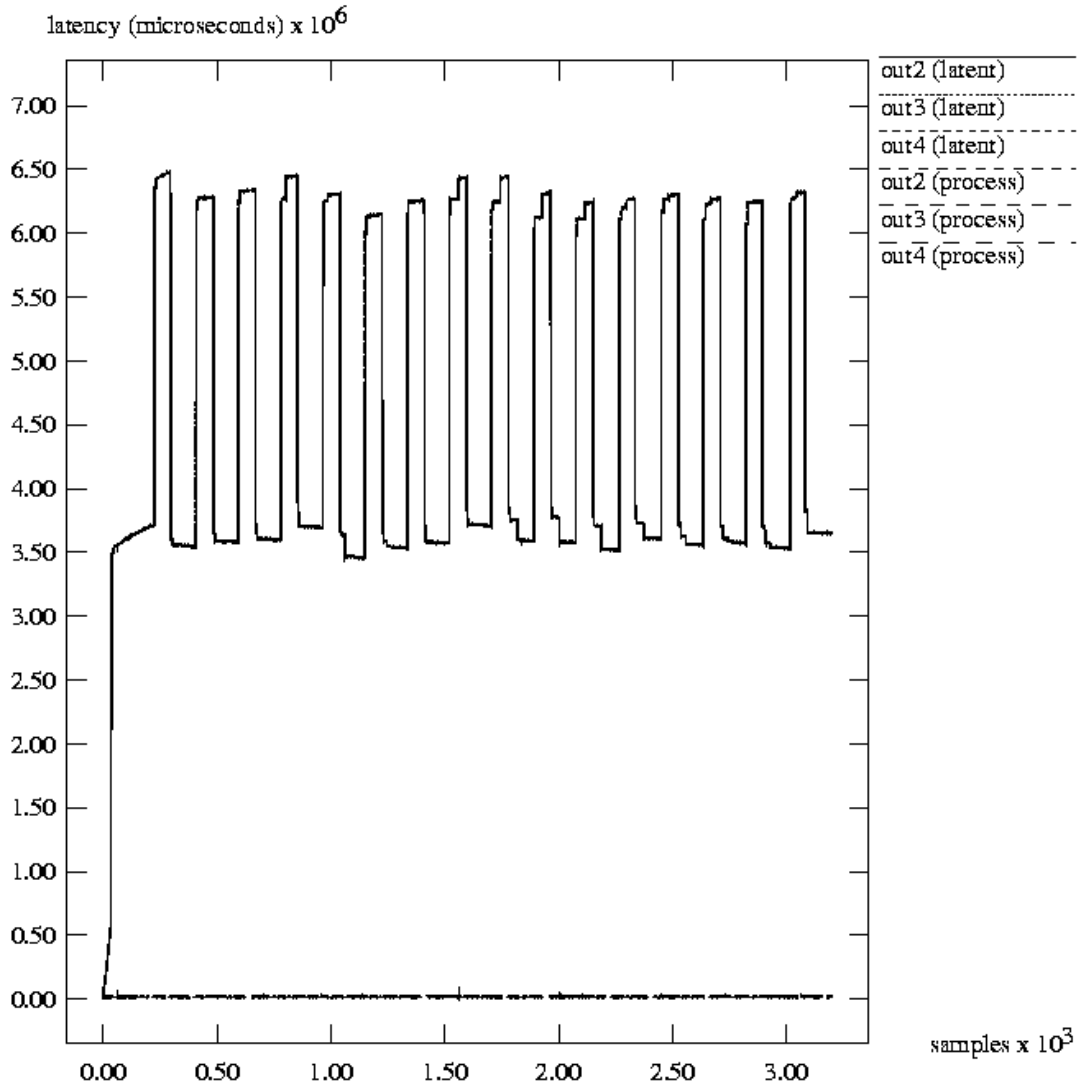
GraphID(008L) drop=300, APE=10000



GraphID(009L) drop=3000, APE=10000

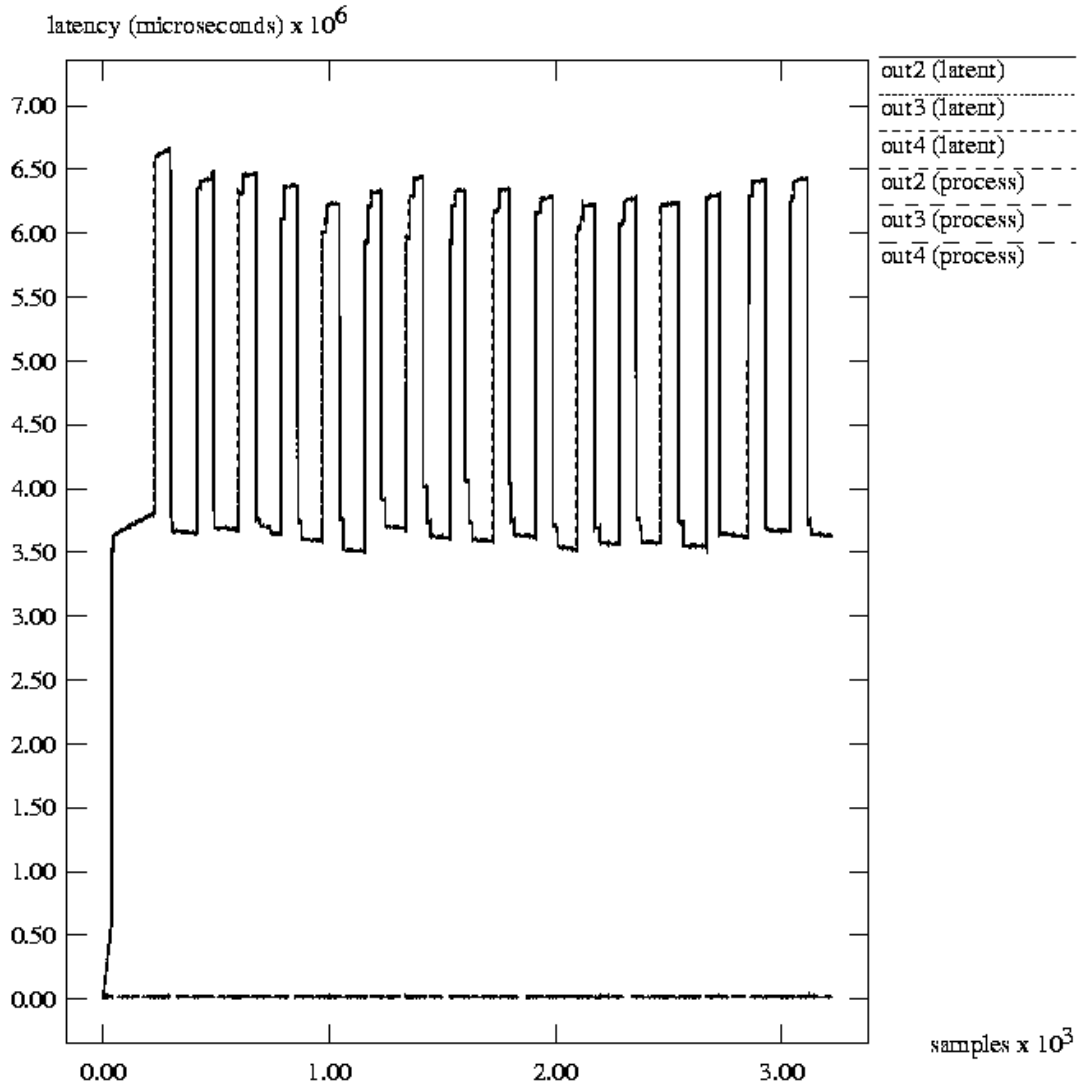


GraphID(010L) drop=30, APE=100000



GraphID(011L) drop=300, APE=100000

The above graph as well as the following graph puzzled me the most the first time I studied them. These graphs show the message latency is 6.5 seconds for some time and then 3.5 seconds repeating over time. These graphs proved that there was a critical issue in DLoVes performance, which challenged me to modify the Worker's algorithm to fix this problem as it is shown further in this chapter.

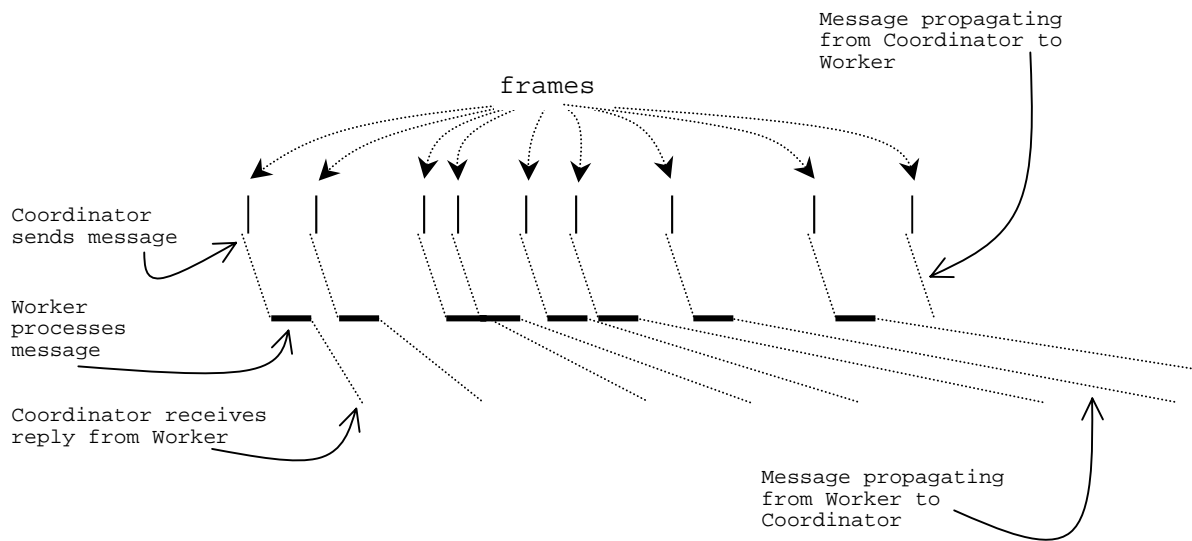


GraphID(012L) drop=3000, APE=100000

These latency graphs show inefficient behavior, because the Coordinator sends many requests to the Workers, overloading the sockets. Thus the Workers have a difficult time sending replies. A new modified algorithm, shown later, resolves this problem.

I hypothesized that the latency of Workers increases due to evaluating older and older requests. The Coordinator continues sending requests to the Workers,

swamping them with requests. The Workers buffer the requests and reply back to the Coordinator in the order in which they receive the requests. After a few seconds, however, Workers start processing older and older requests, and as a result, sockets become congested. The Coordinator may use a high frame rate, but the frames become more and more out-of-date.



Workers drifting in responding back to Coordinator

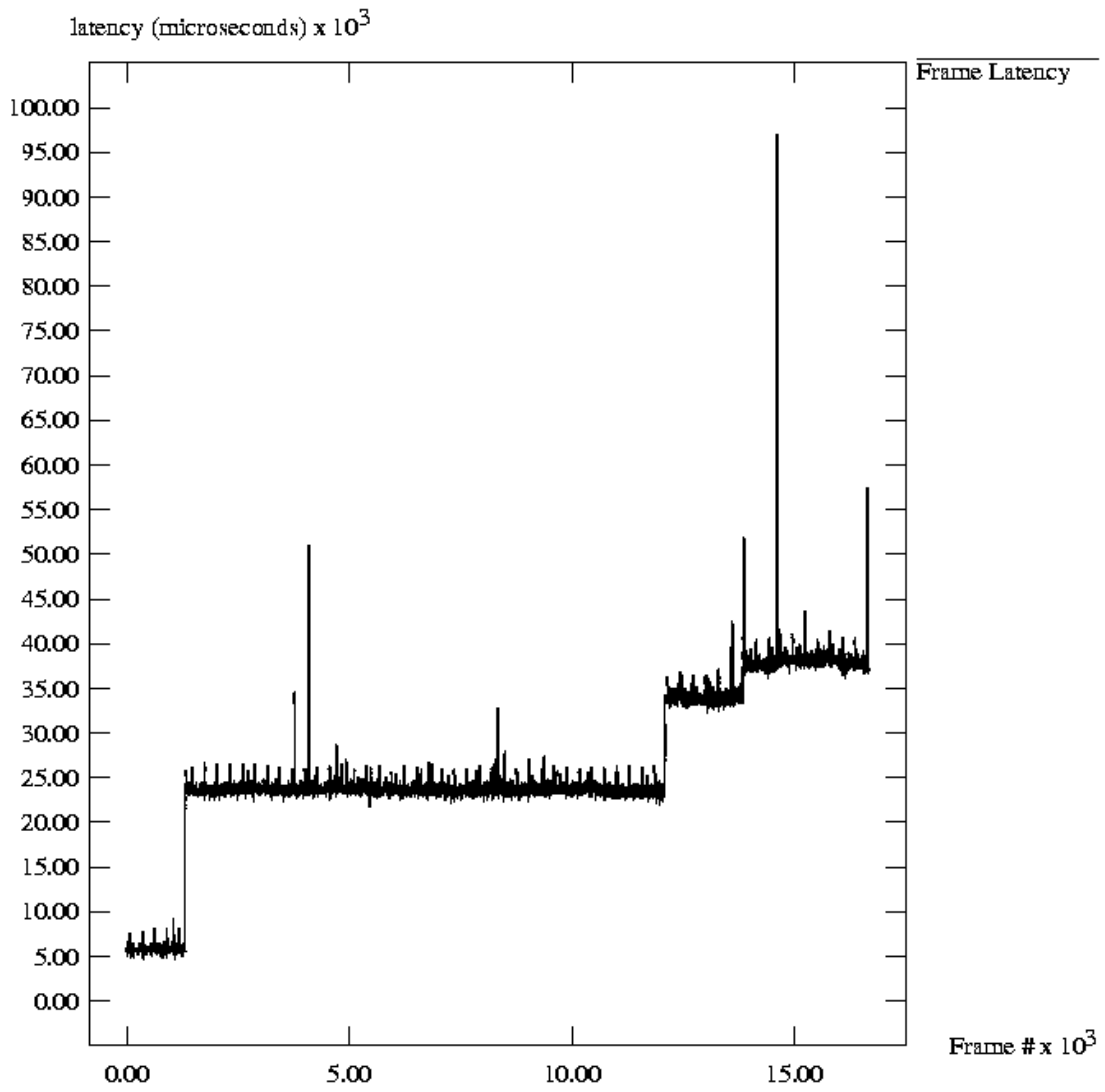
The Coordinator sends requests to all three Workers so that replies come from all three Workers, adding much network traffic. We seem to observe a step increase in latency every time there is an increase in congestion. The higher the latency, the older the requests the Workers are processing.

Frame Validity and Statistical Skew

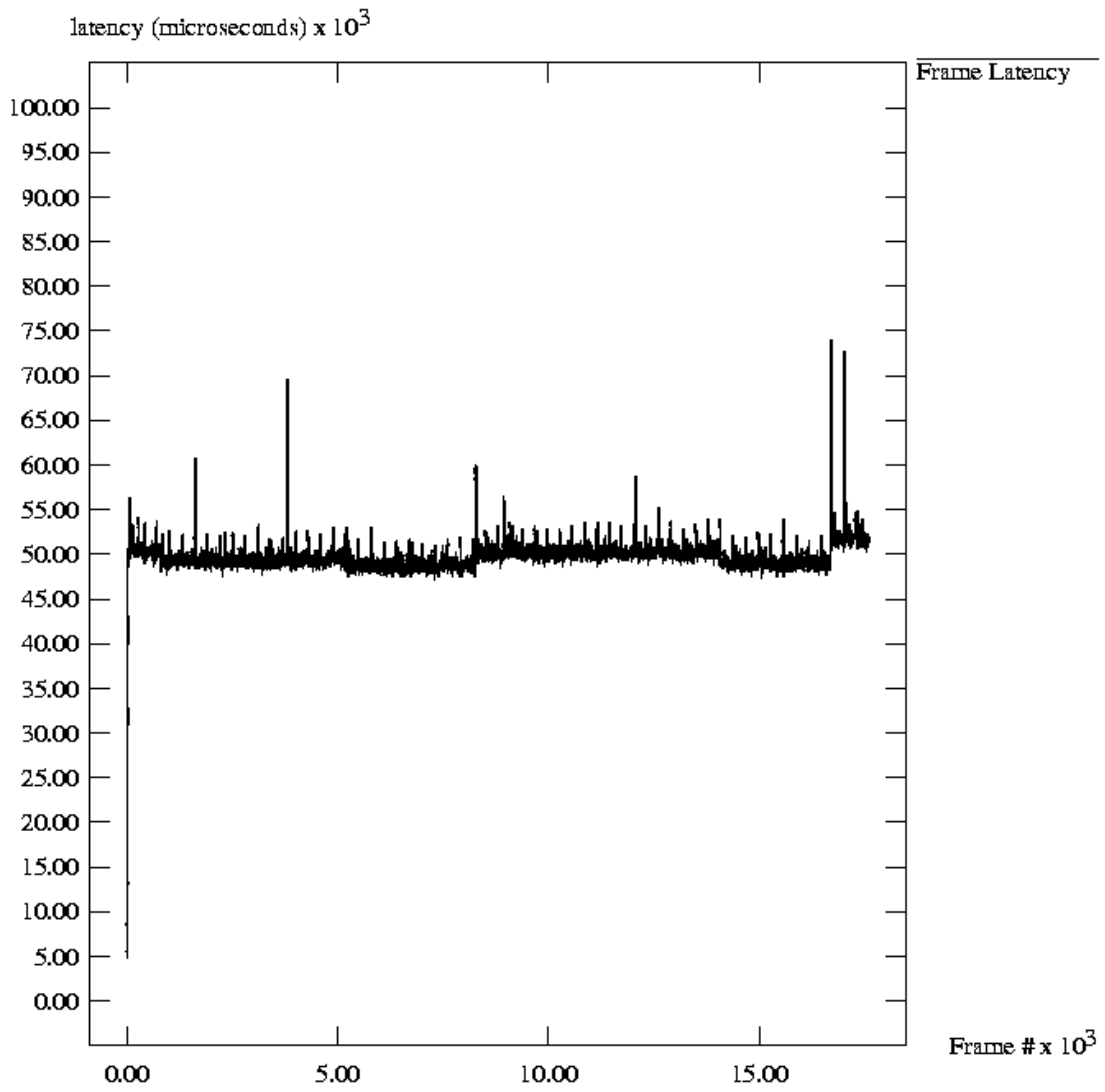
Latency shows how long a message spends on the network to get to the Workers and then come back to Coordinator (time required by a Worker to evaluate a Link is not counted). However, this does not show how accurate the frame rendering is. To visualize how valid the frames are, I used a statistical clock skew as show below:

$$\text{skew} = (\text{wall clock}) - \min(\text{time of request of all Variables})$$

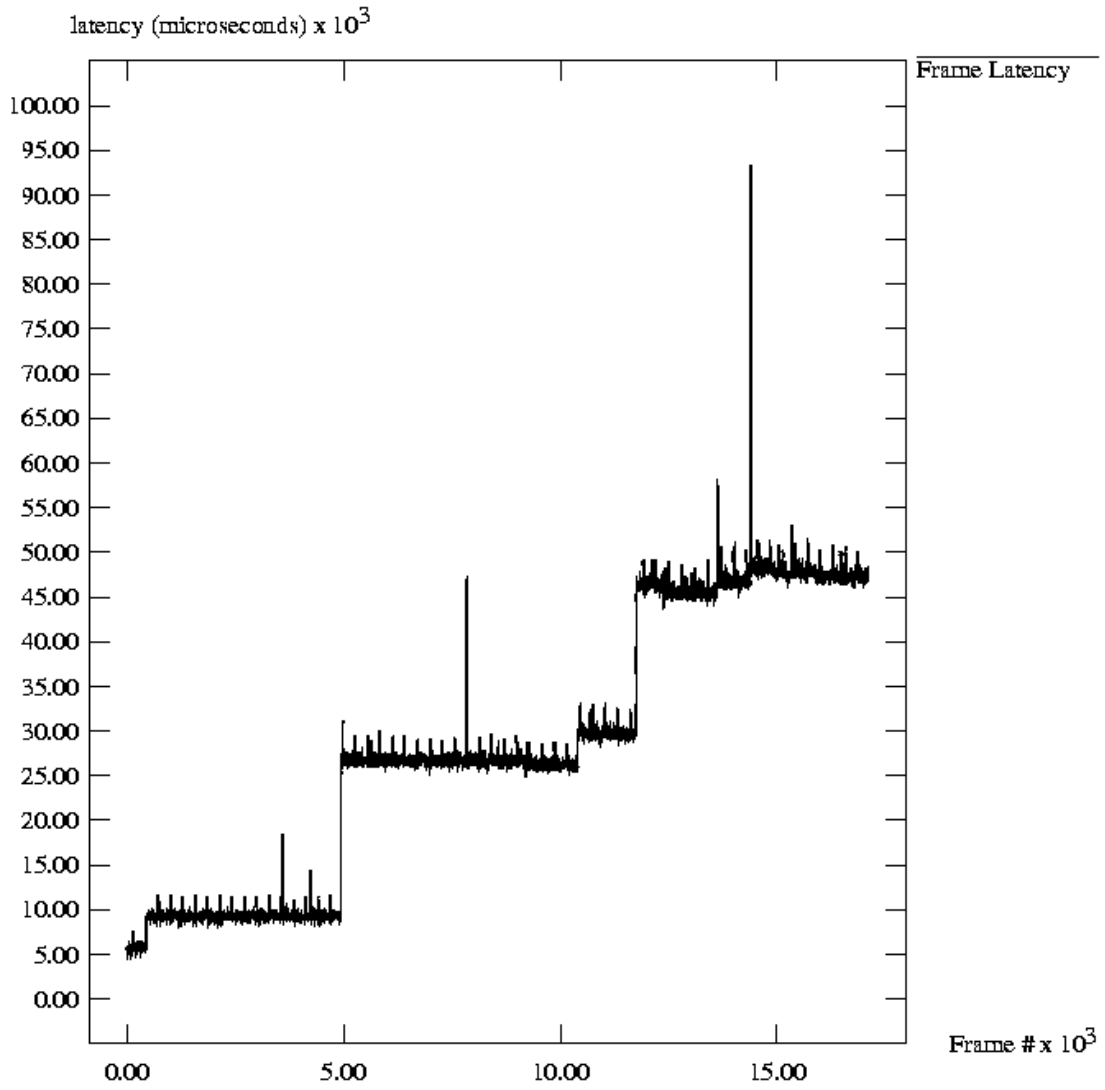
For every frame, the minimum time of request of all output Variables is subtracted from the current time (wall clock). This skew describes the worst difference between what is shown and what the user is doing. I plotted graphs that show this skew over time. The following 12 graphs show how out-of-date each frame is. Let us call the set of Variables that the Coordinator uses to render the display “render”. Every time a request comes back from a Worker it is time-stamped with the current time indicating the time the Variable is lastly updated. When the Coordinator is about to render the display it gets the system’s clock and subtracts from it the minimum time of the Variables in the set “render”.



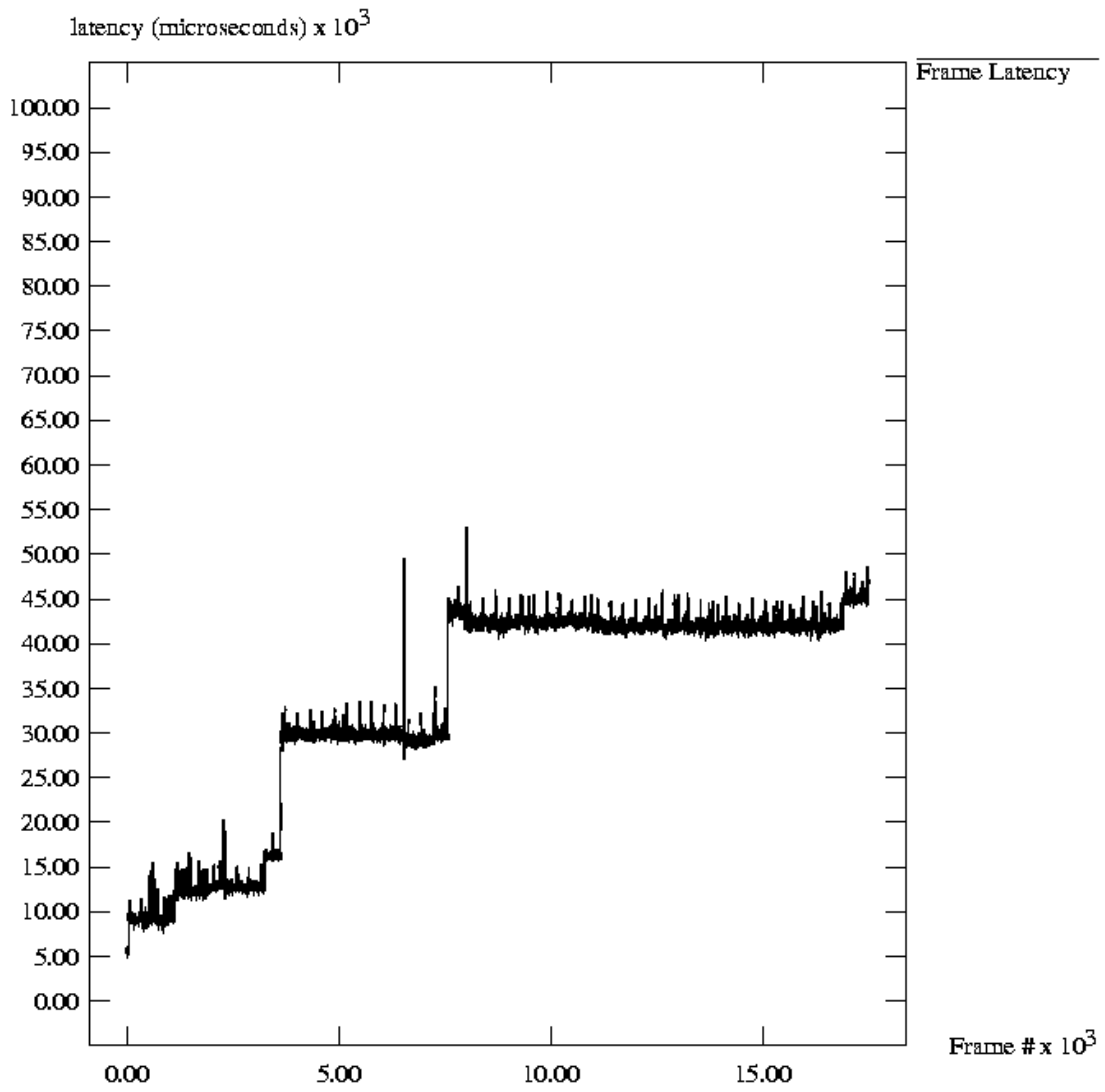
GraphID(001F) drop=30, APE=100



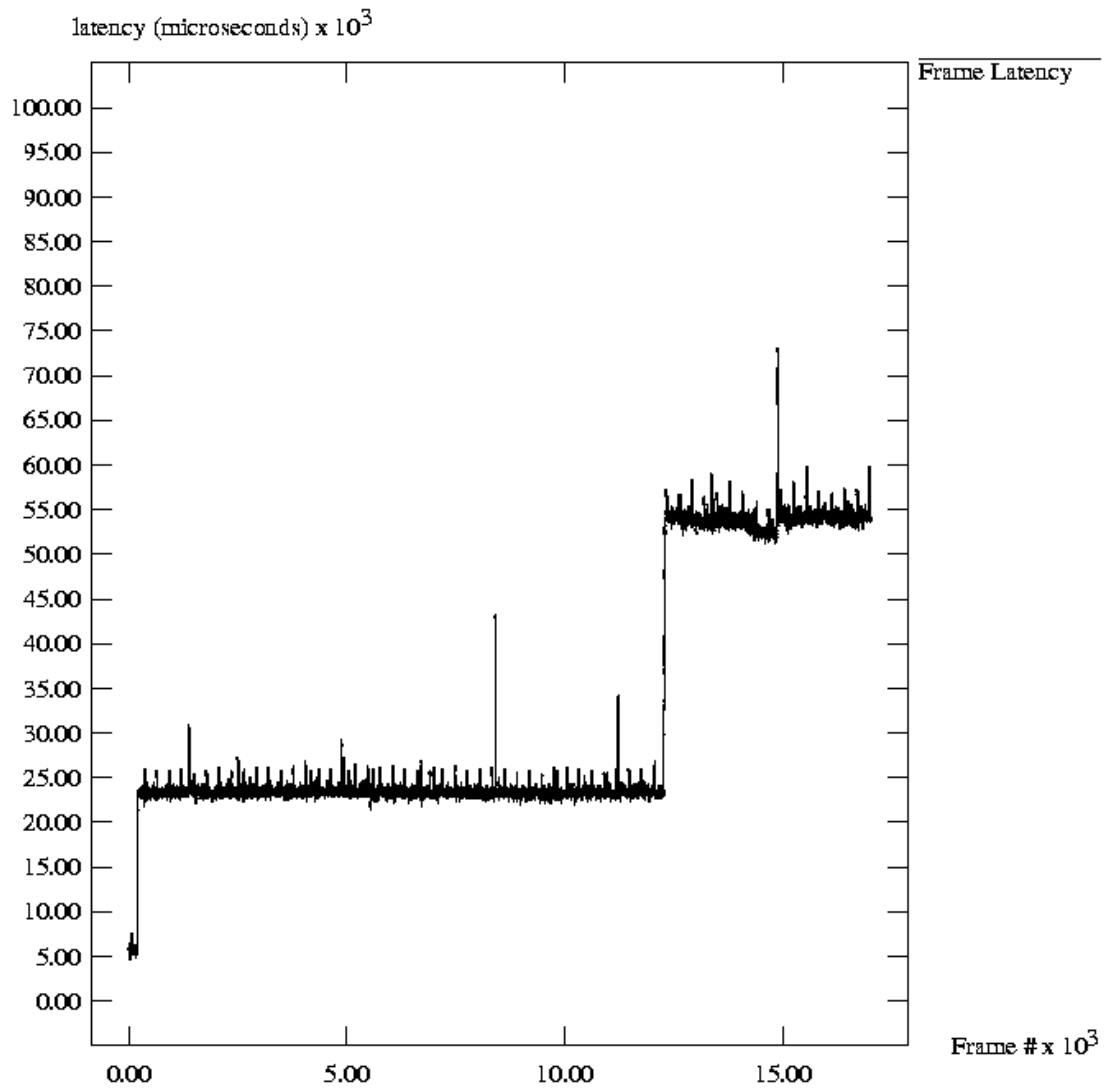
GraphID(002F) drop=300, APE=100



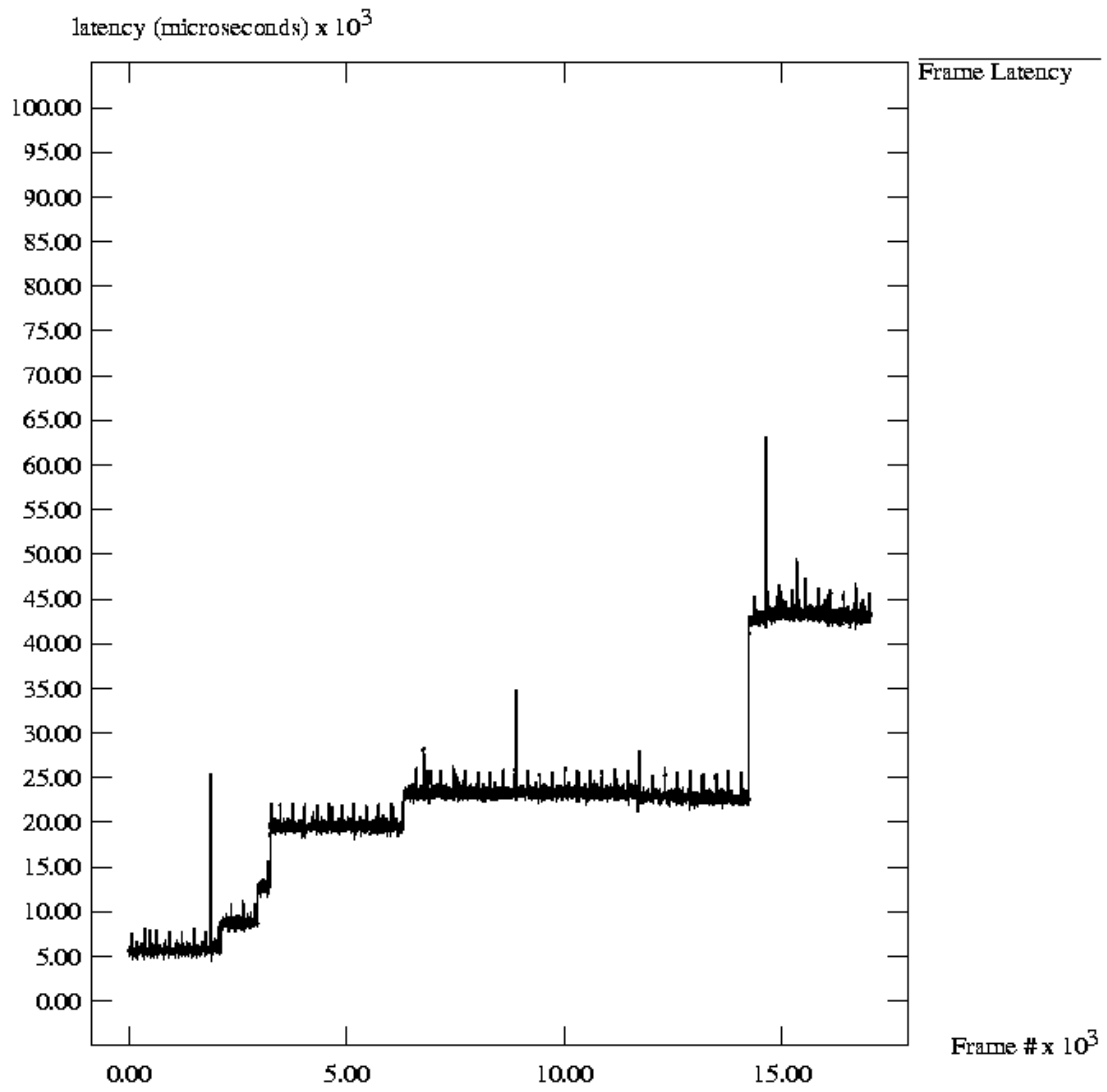
GraphID(003F) drop=3000, APE=100



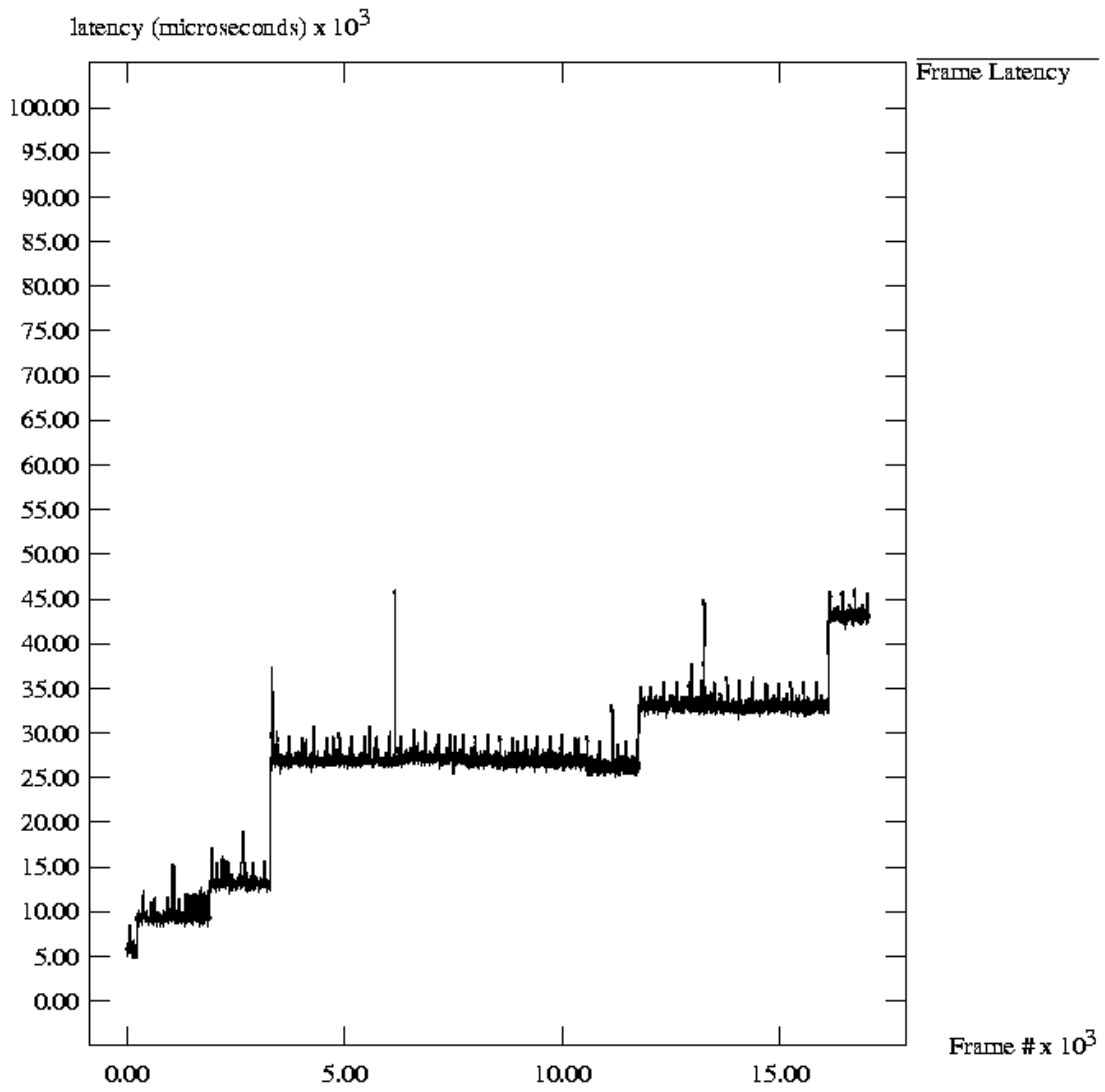
GraphID(004F) drop=30, APE=1000



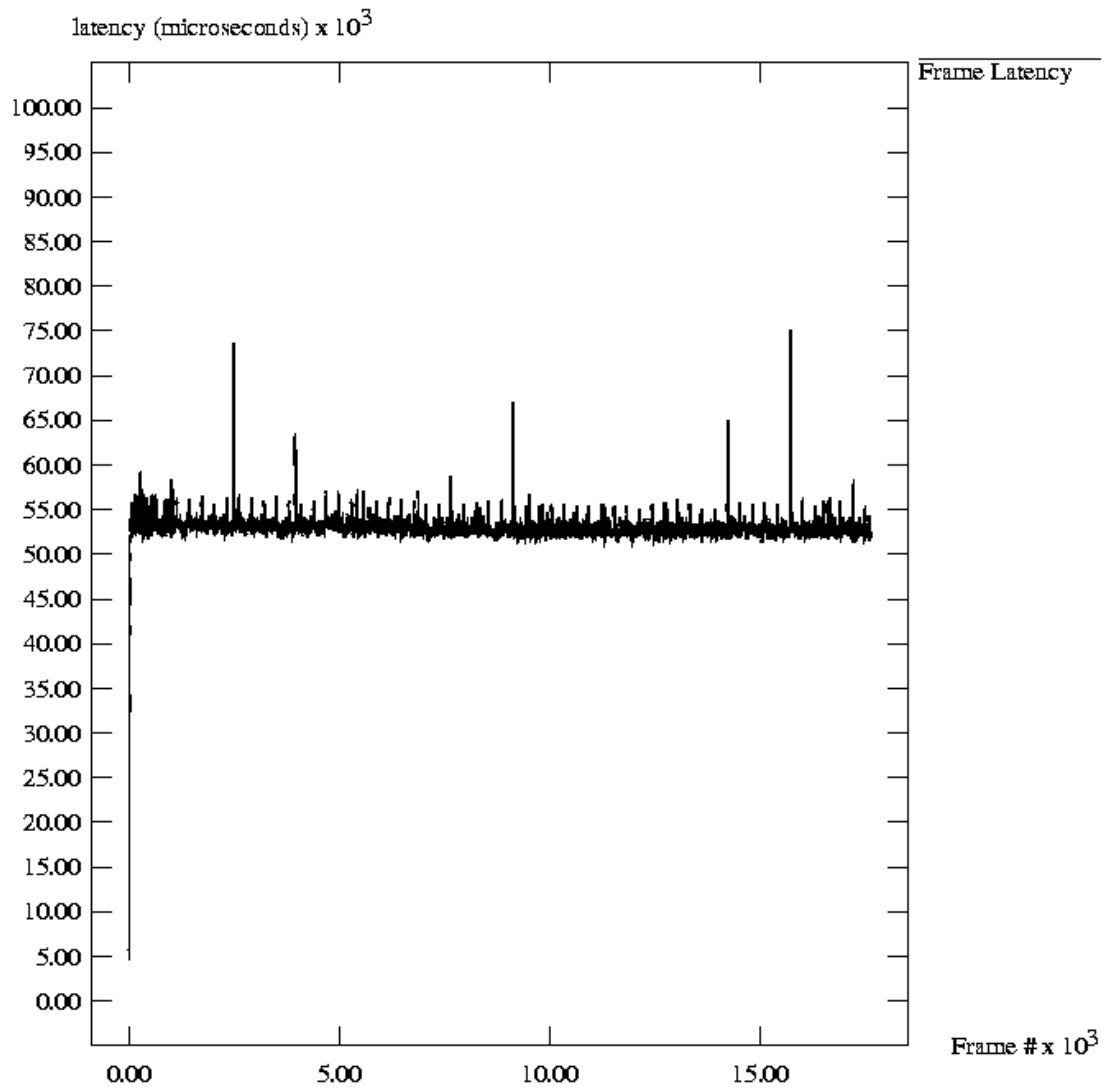
GraphID(005F) drop=300, APE=1000



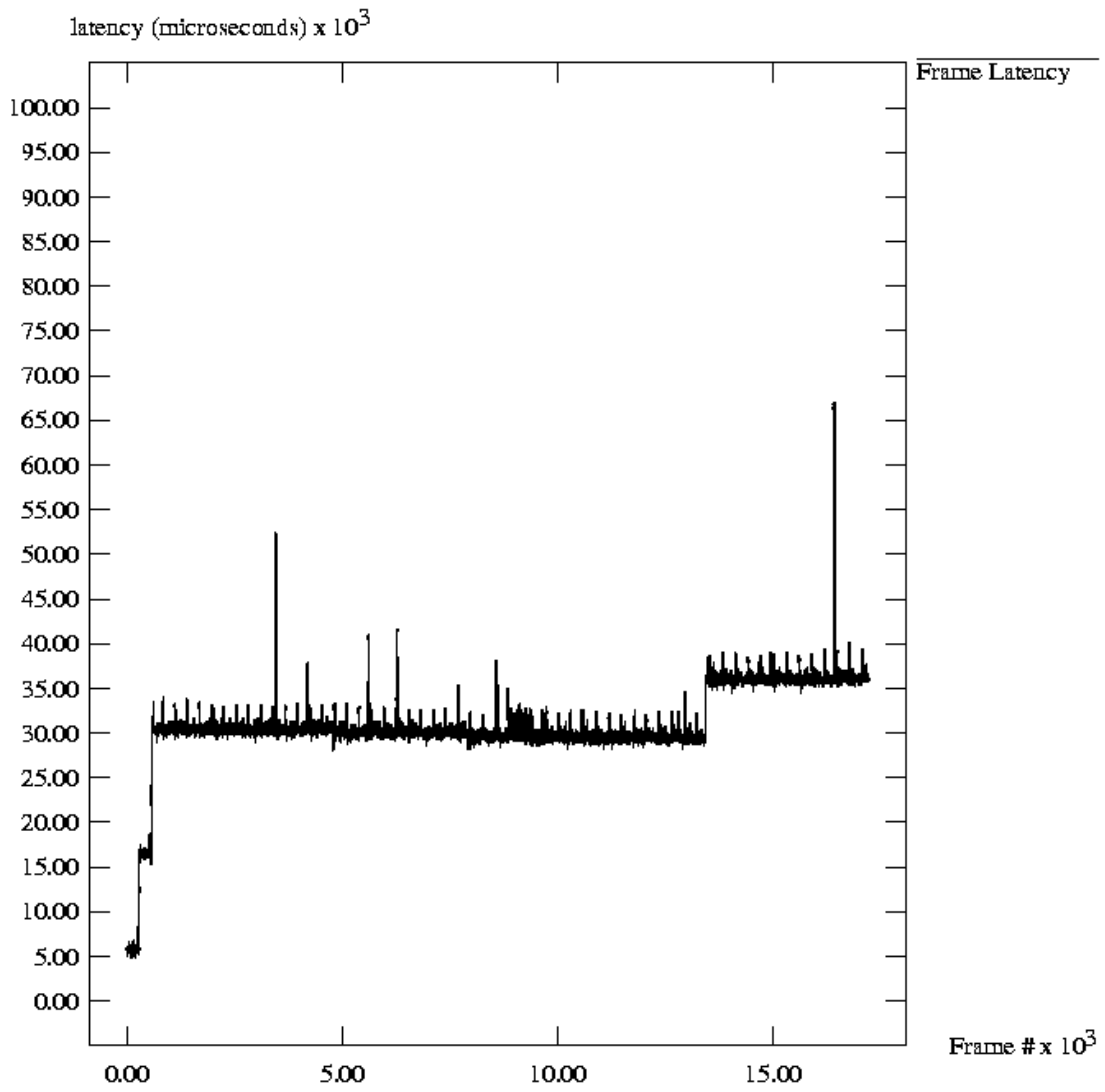
GraphID(006F) drop=3000, APE=1000



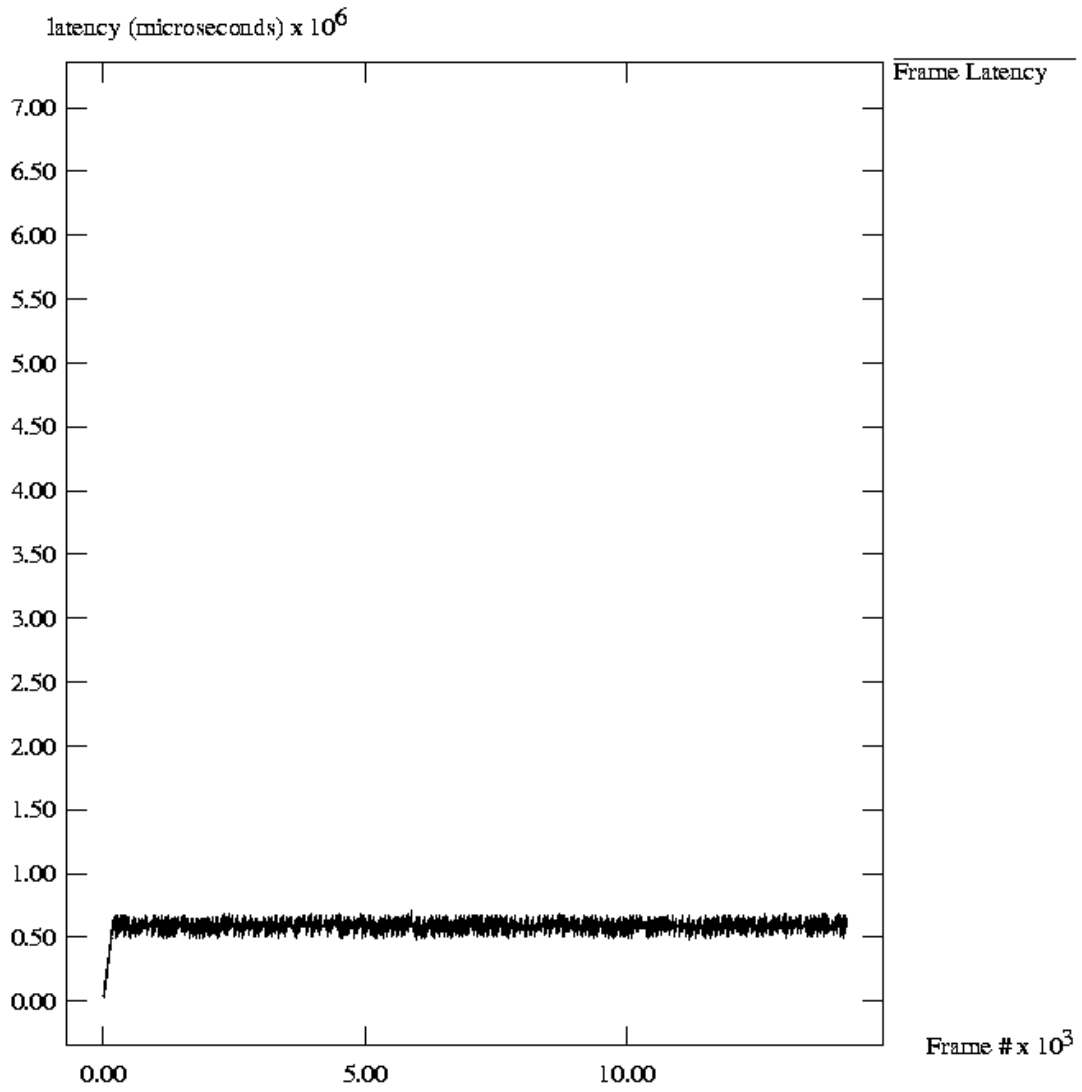
GraphID(007F) drop=30, APE=10000



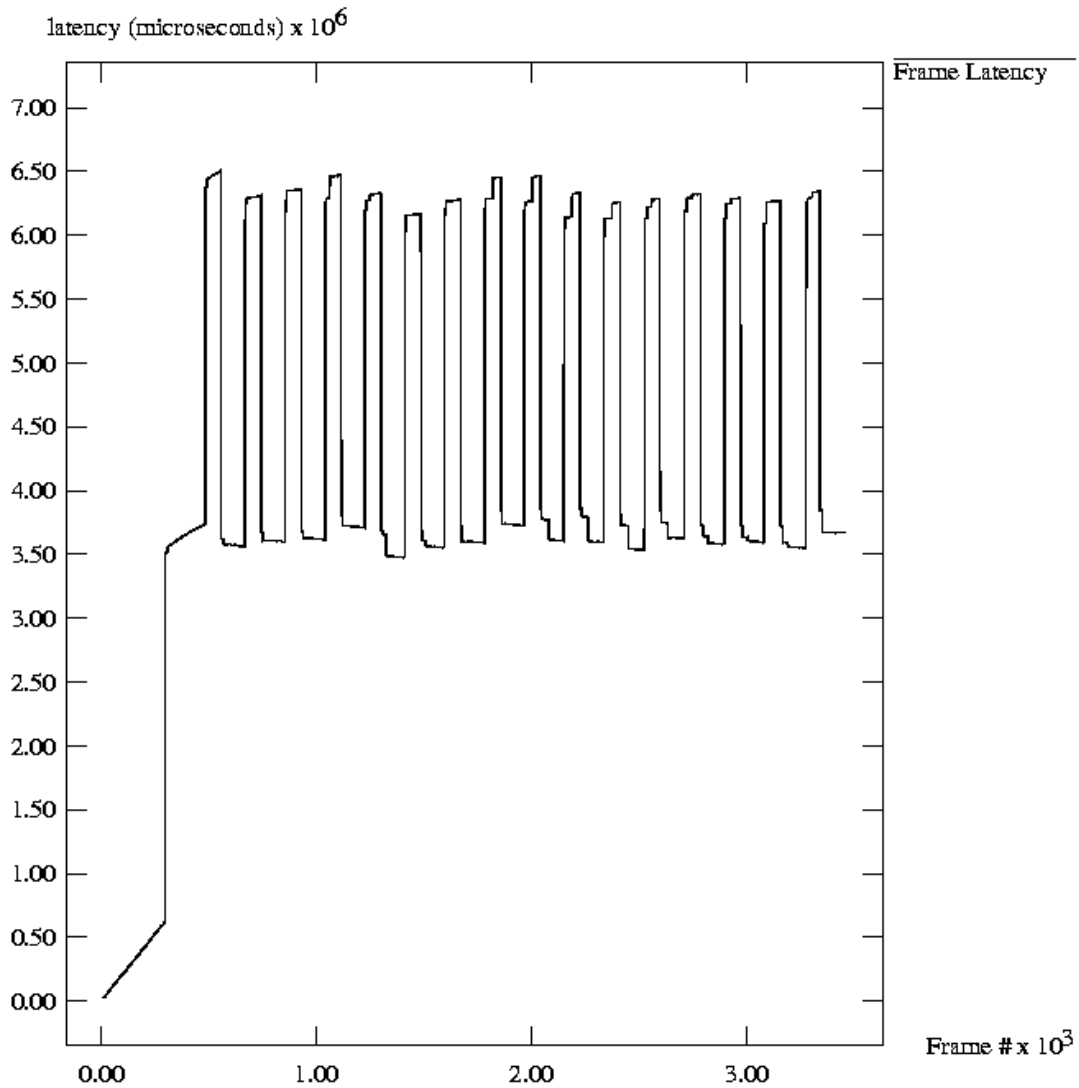
GraphID(008F) drop=300, APE=10000



GraphID(009F) drop=3000, APE=10000

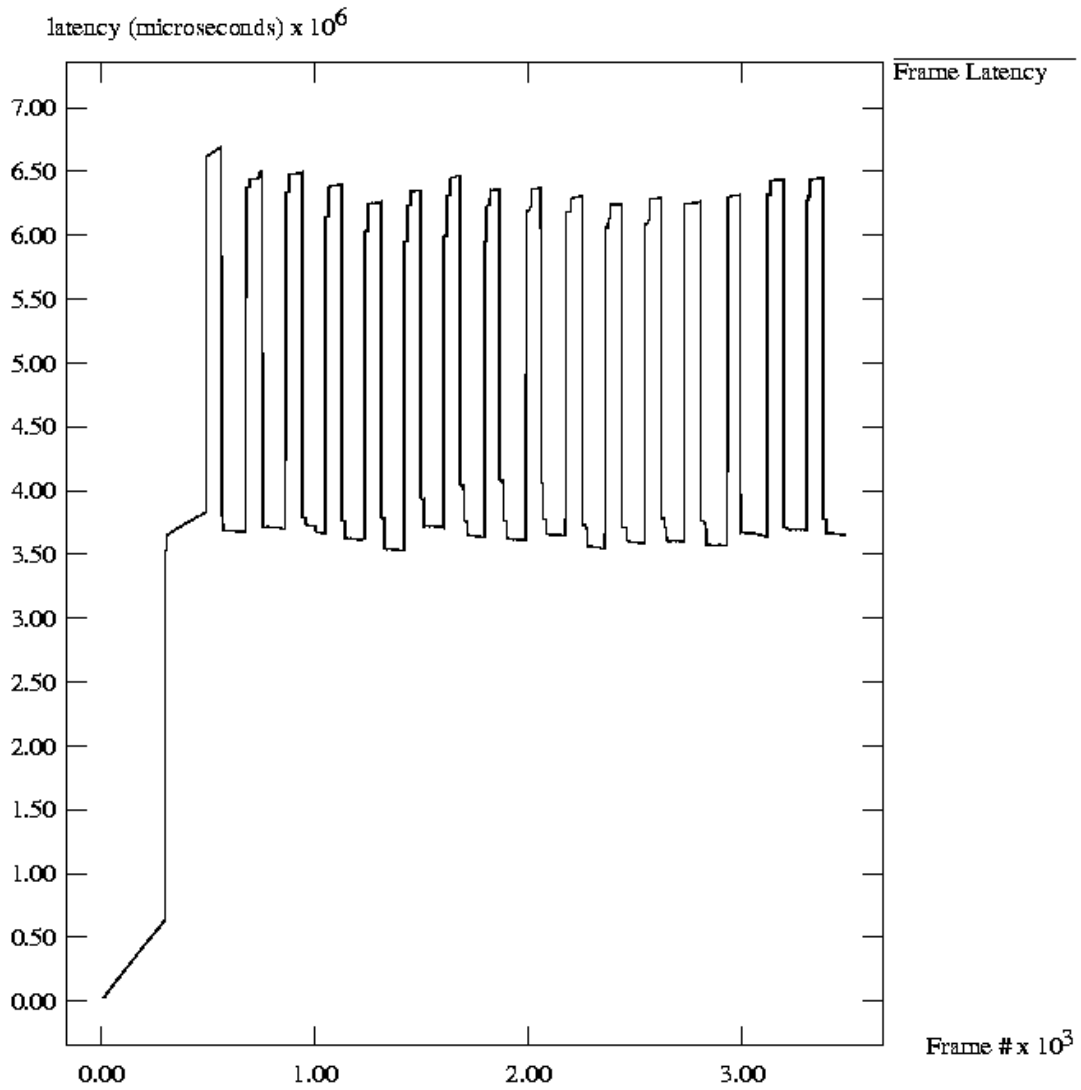


GraphID(010F) drop=30, APE=100000



GraphID(011F) drop=300, APE=100000

The above graph shows that in the best case the Coordinator renders the display with information that is 3.5 seconds old. This is expected when we study the message latency graph of this experiment (*GraphID(001L)*). Because all messages are either 3.5 seconds old or 6.5 seconds old, the Coordinator uses Variables that were requested so many seconds ago, thus frame accuracy graphs follow message latency graphs patterns.

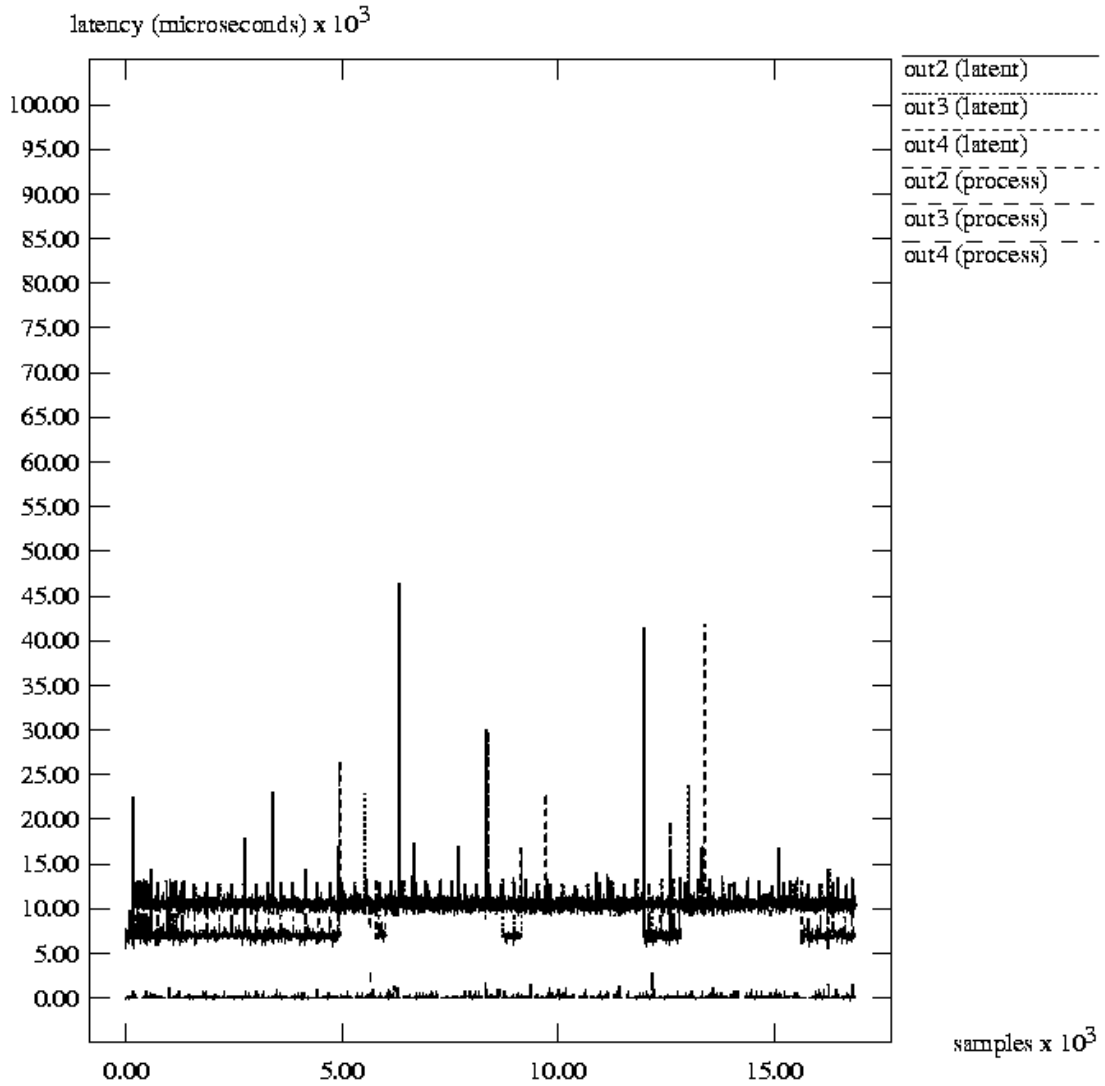


GraphID(012F) drop=3000, APE=100000

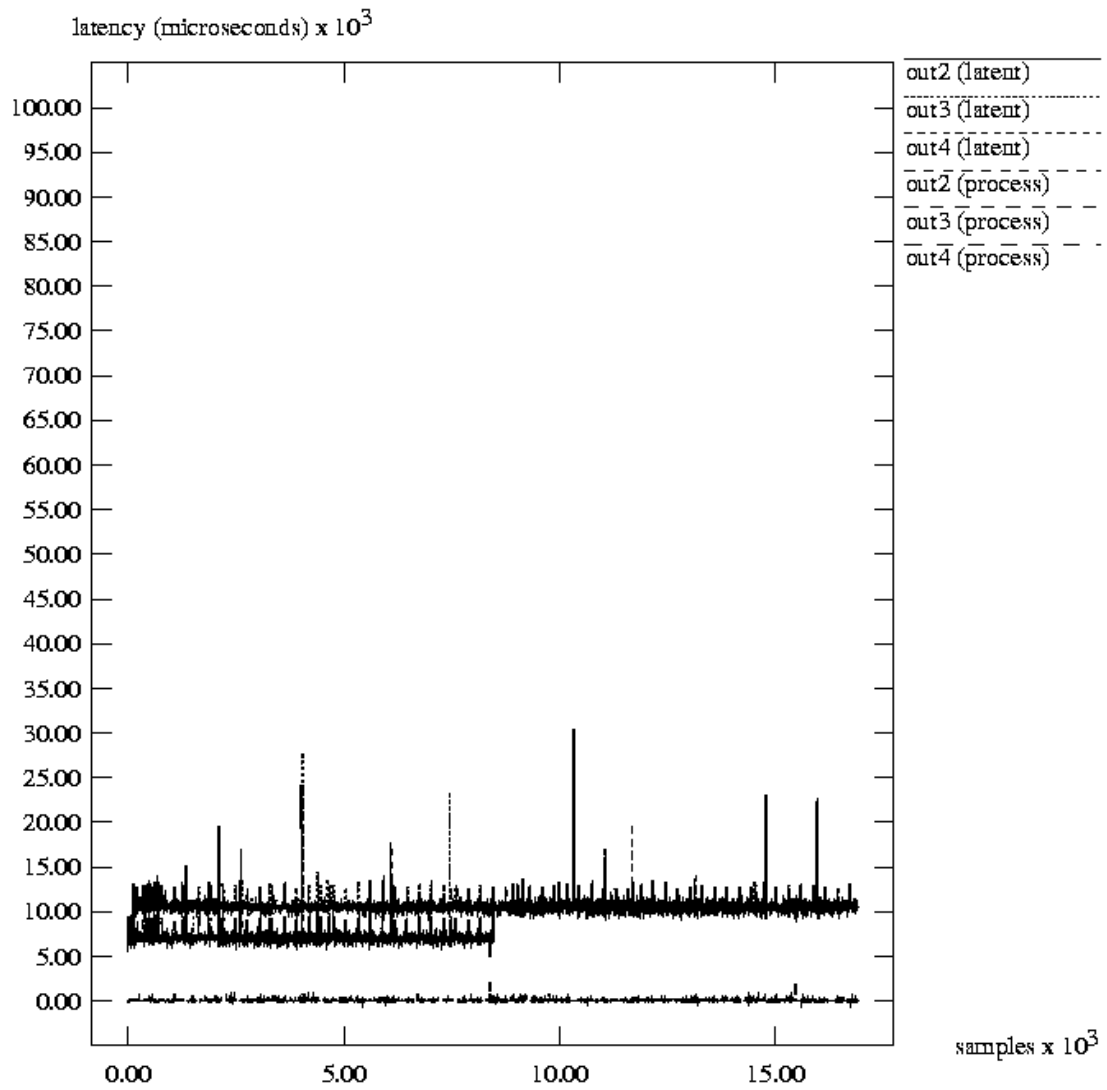
Modified Algorithm for the Workers

To overcome the problem of the stepwise increase in both latency and frame accuracy graphs, I modified the algorithm of the Workers to throw away any old

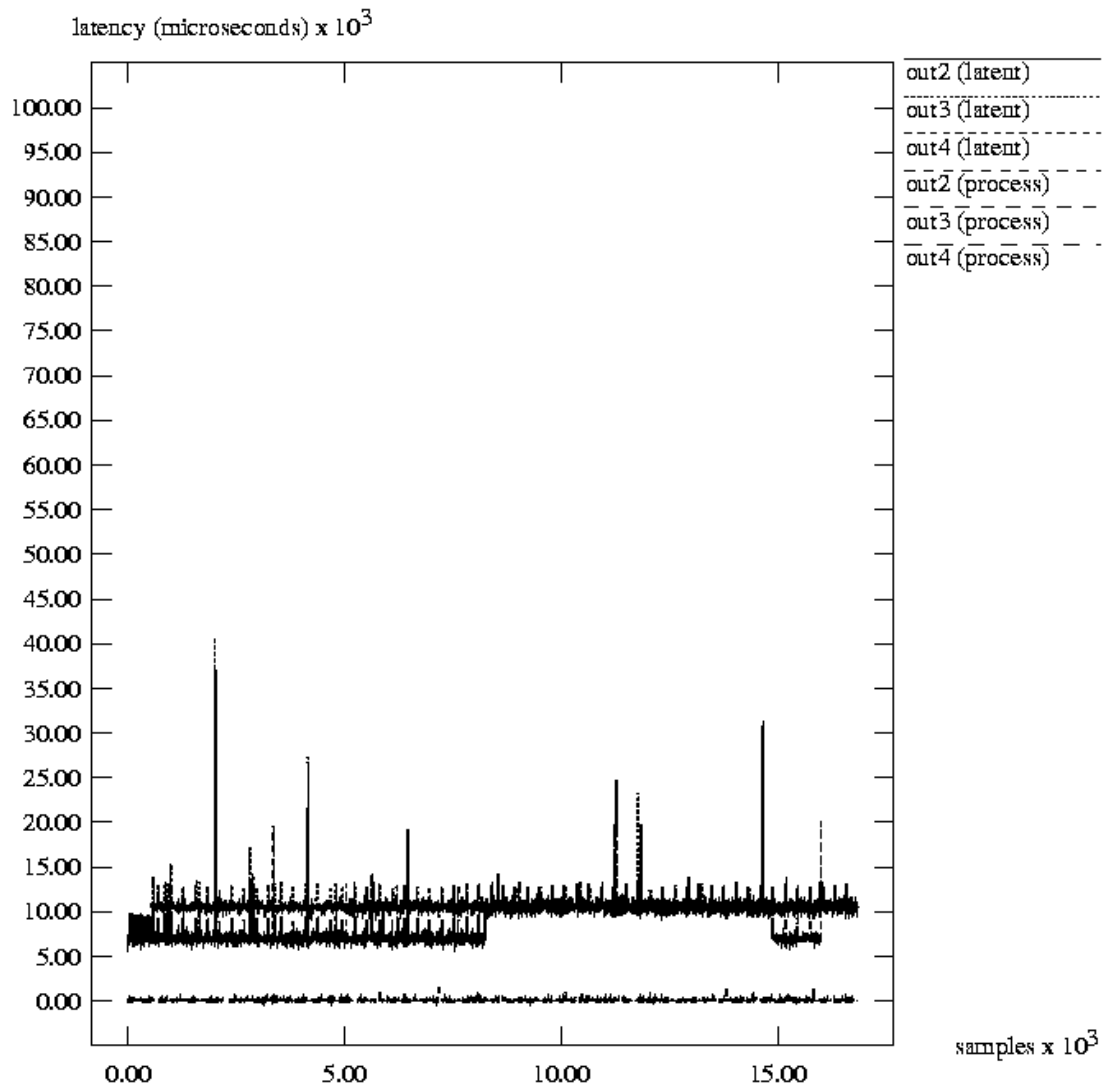
requests in the queue and only evaluate the most recent ones. The following graphs show latency using this modified version of the Workers' algorithm.



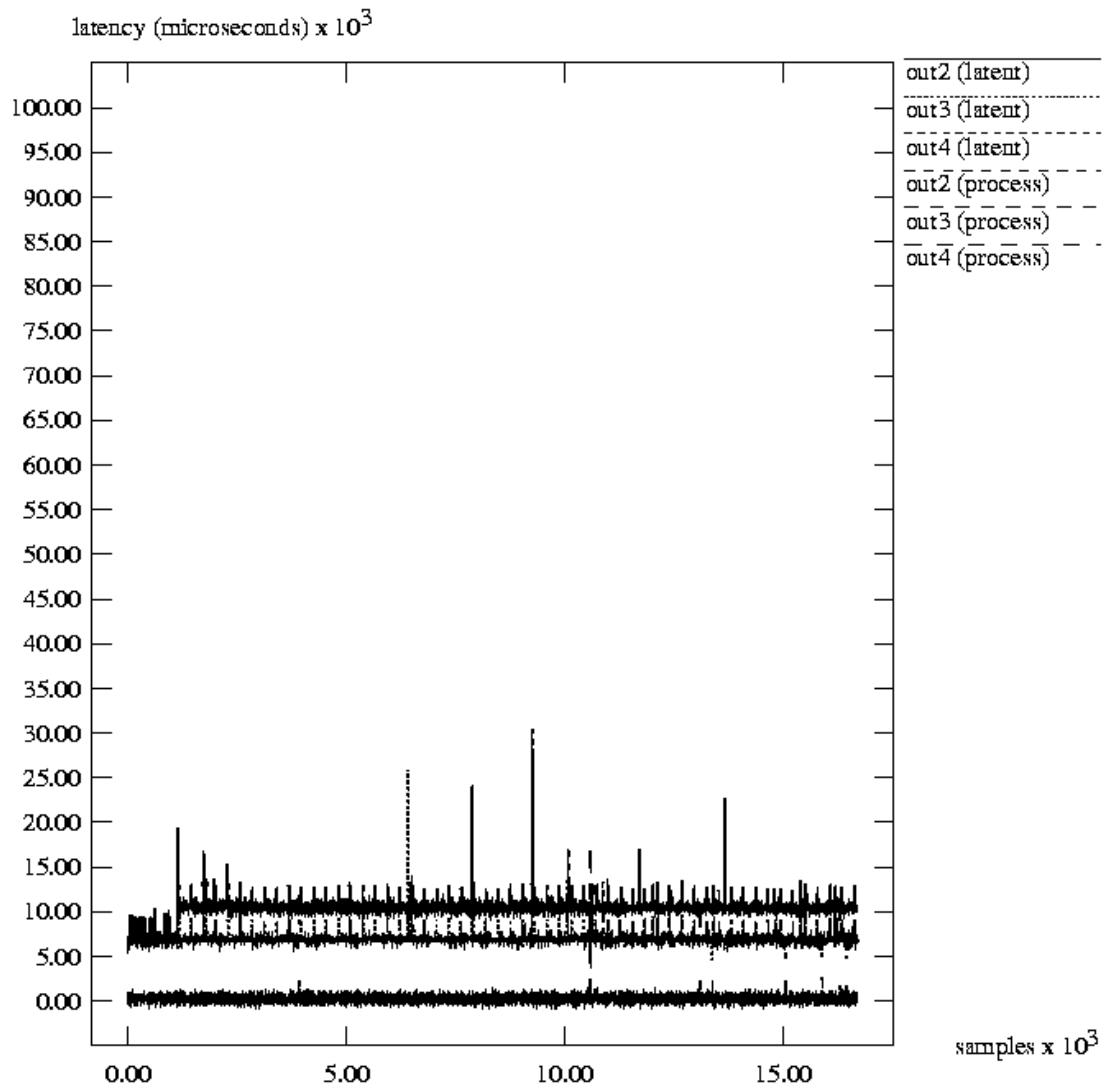
GraphID(101L) drop=30, APE=100



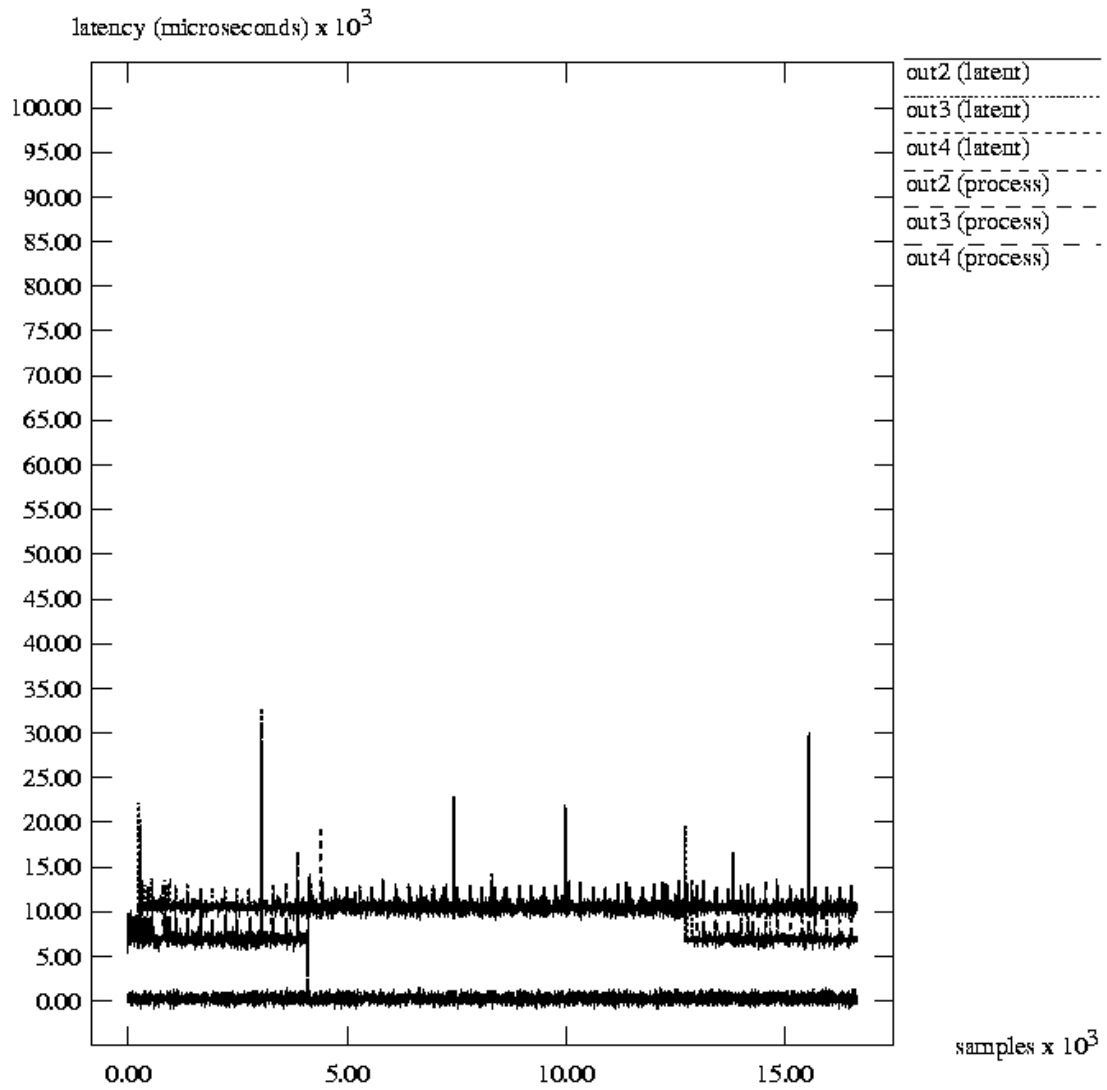
GraphID(102L) drop=300, APE=100



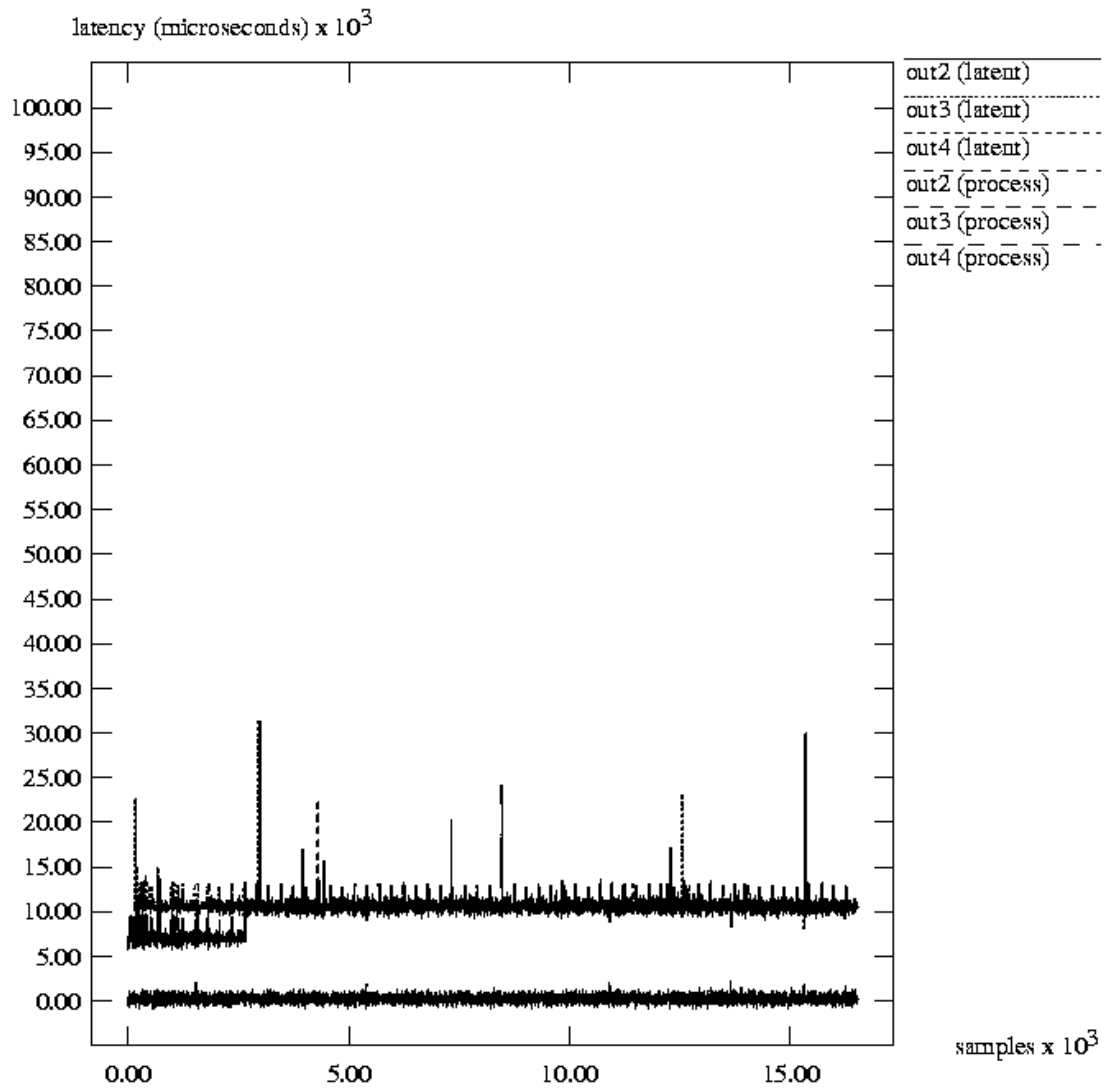
GraphID(103L) drop=3000, APE=100



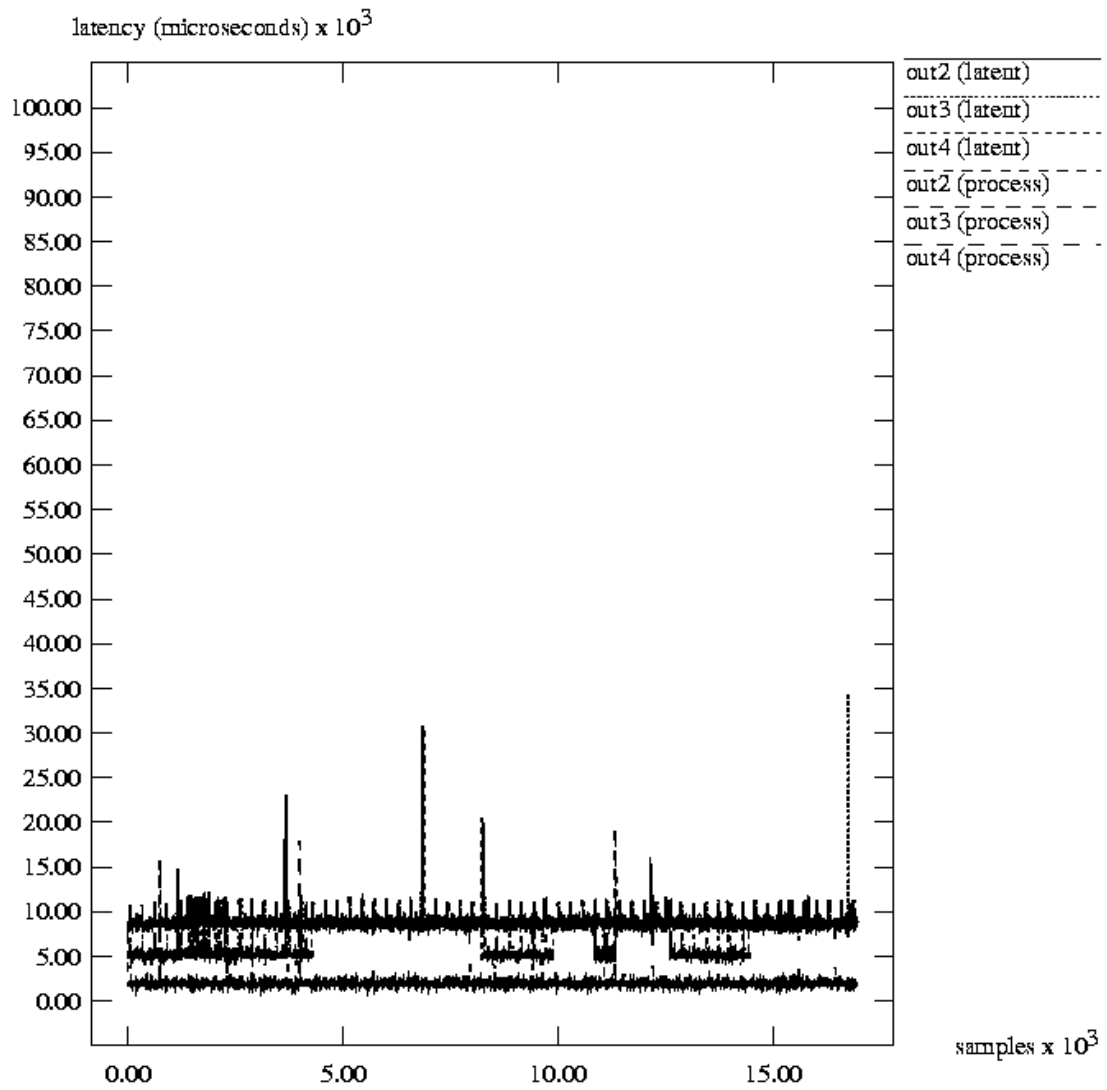
GraphID(104L) drop=30, APE=1000



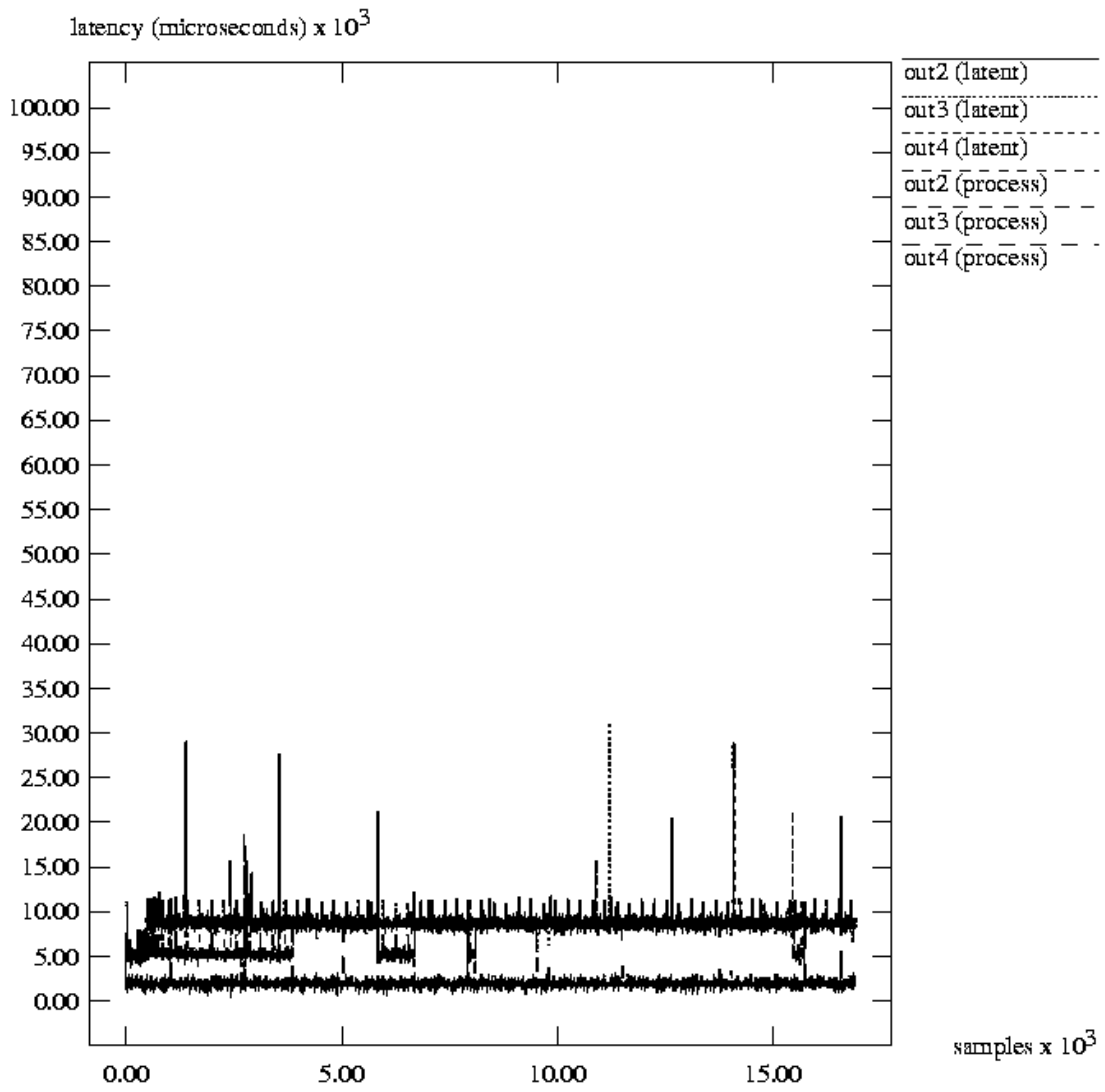
GraphID(105L) drop=300, APE=1000



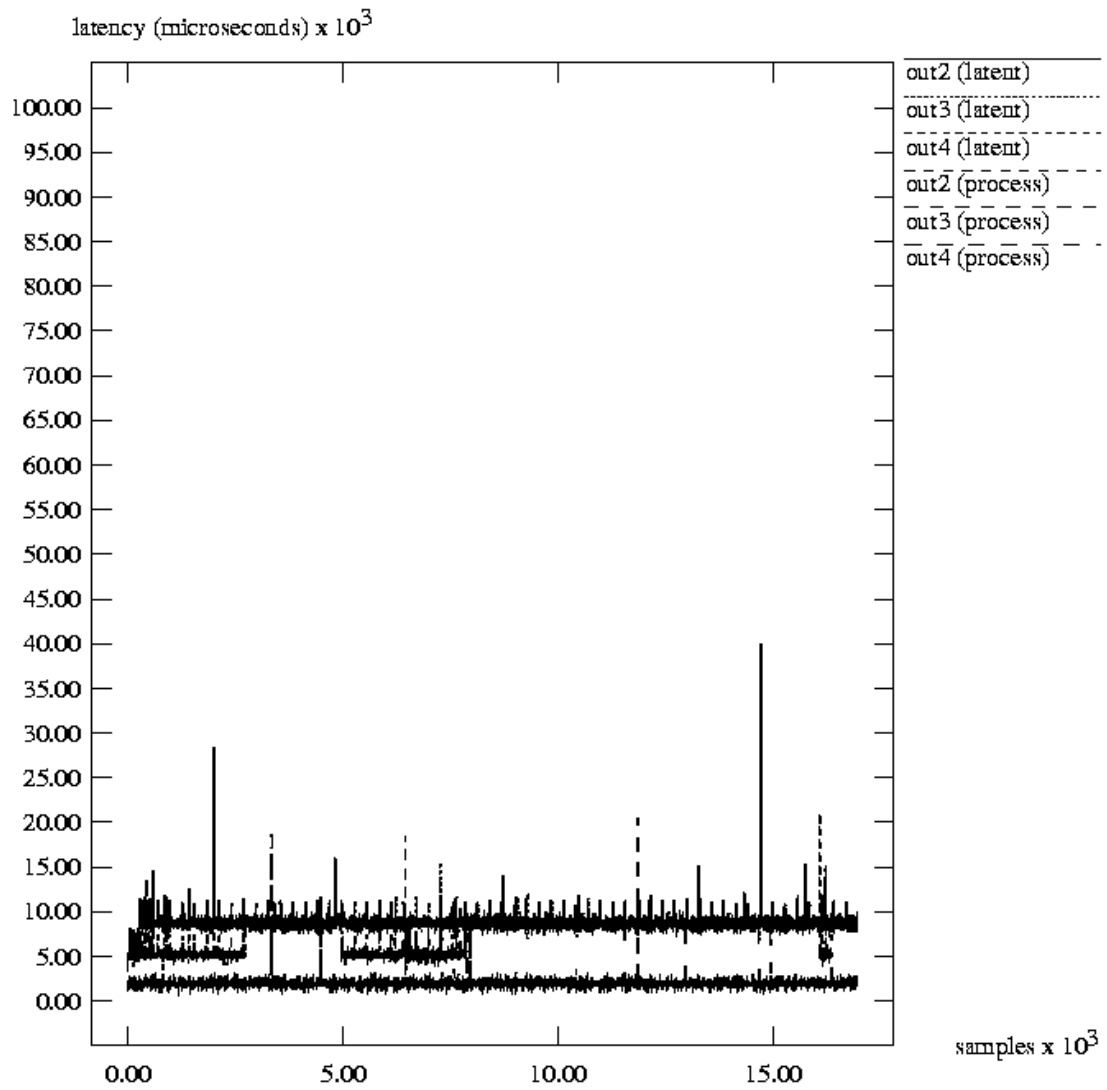
GraphID(106L) drop=3000, APE=1000



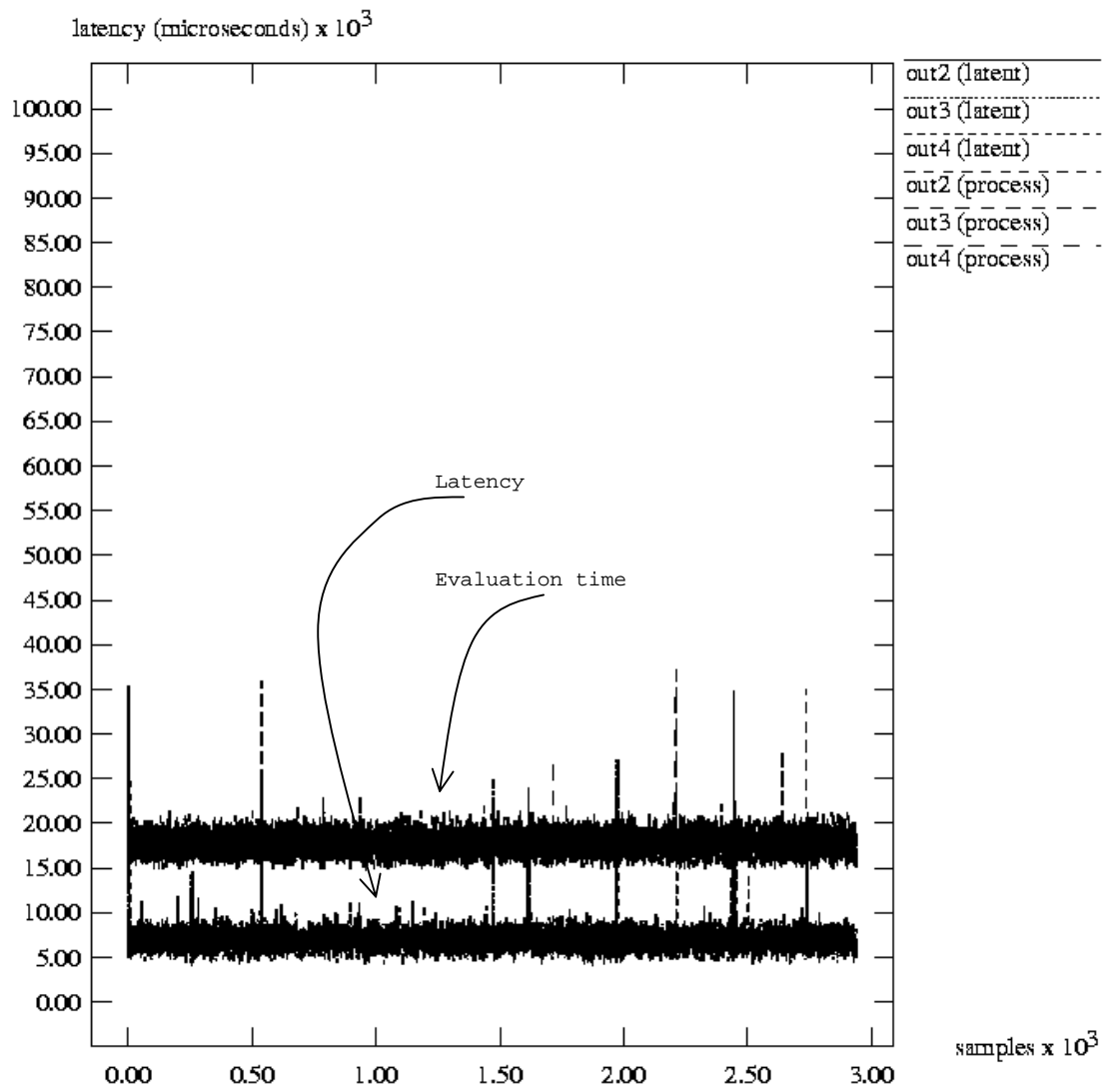
GraphID(107L) drop=30, APE=10000



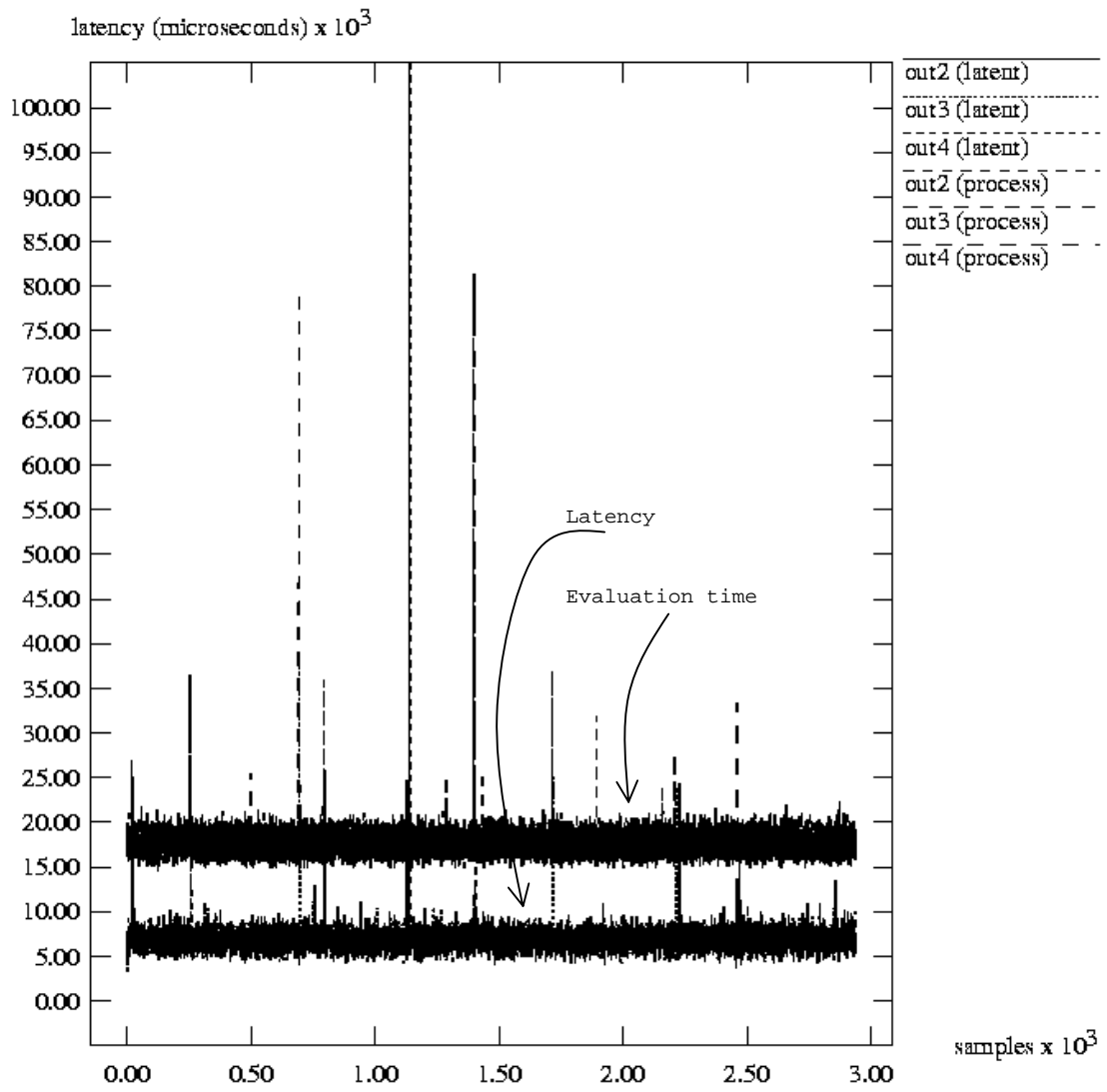
GraphID(108L) drop=300, APE=10000



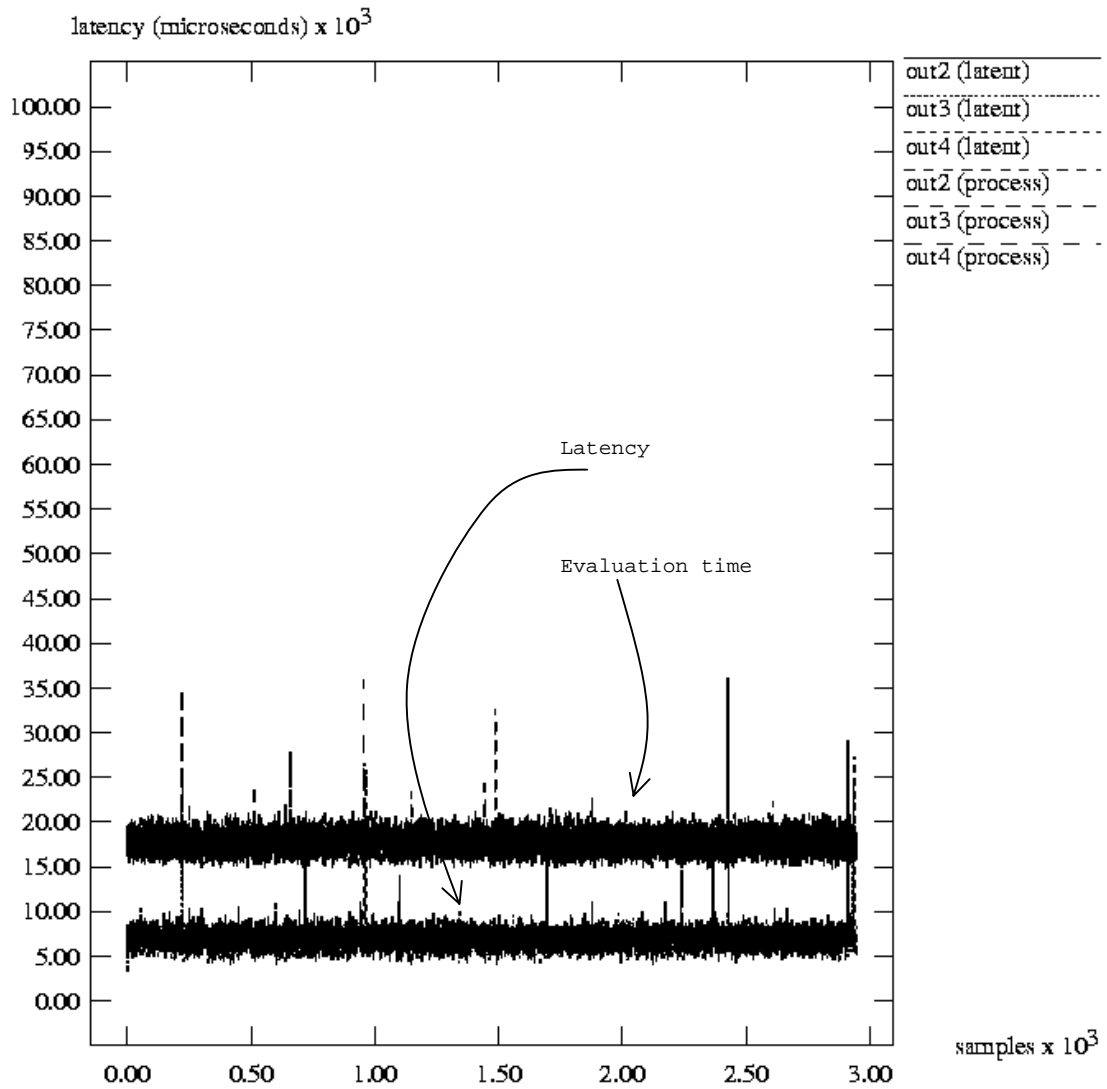
GraphID(109L) drop=3000, APE=10000



GraphID(110L) drop=30, APE=100000



GraphID(111L) drop=300, APE=100000

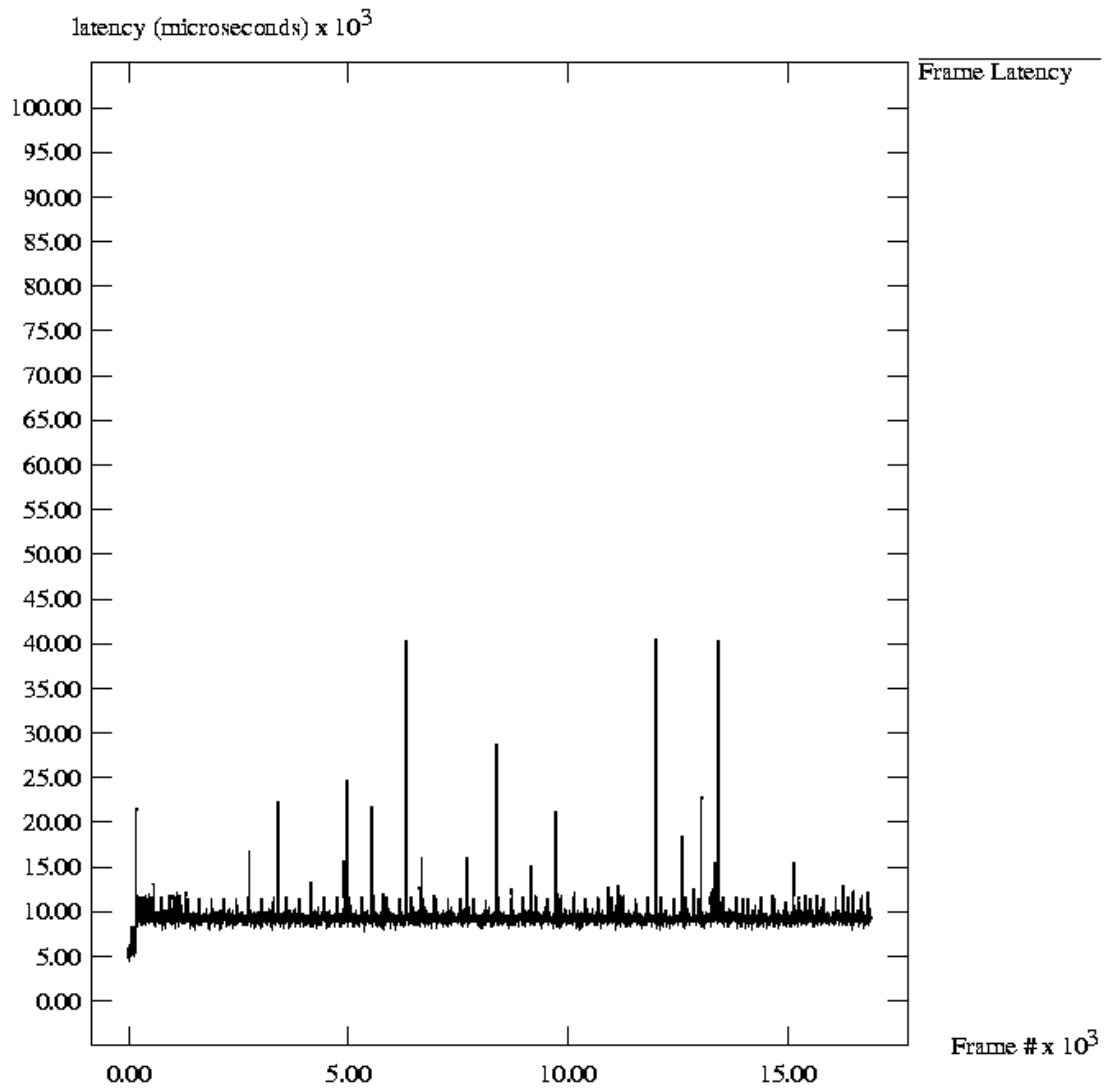


GraphID(112L) drop=3000, APE=100000

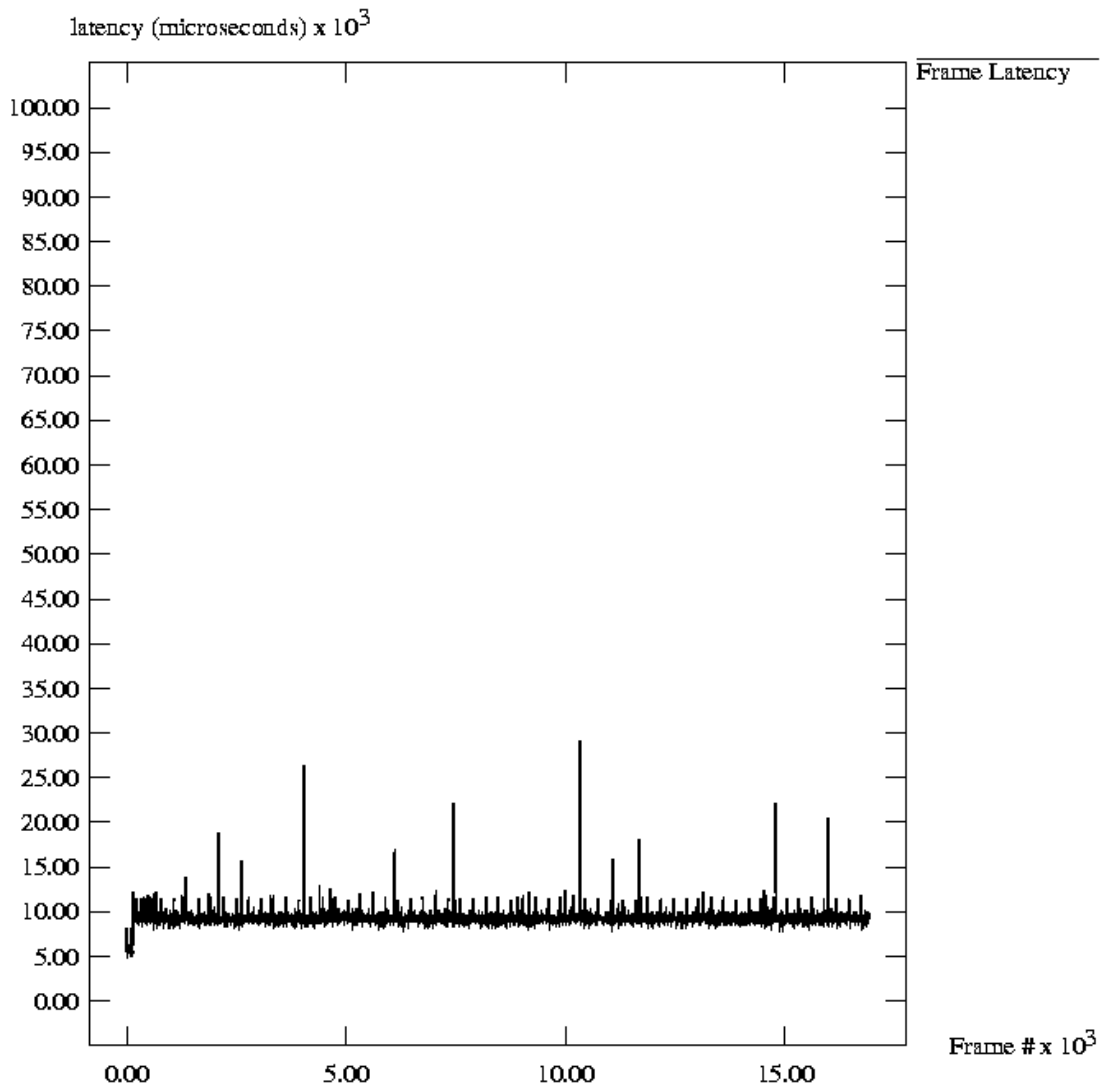
This new modified algorithm has a dramatic effect on the accuracy of the frames. The accuracy stays the same throughout the experiment. This algorithm allows the Coordinator to use the most up-to-date Variables for rendering because messages are sent back and fourth very quickly as a result of a decrease in network congestion.

In the above 12 graphs, the abrupt increase in the latency disappeared. This is because the Workers only process the most recent requests, disregarding any older requests (and this is okay since the transactions are performed in a streaming manner as it was shown earlier). Using the modified version of the Workers' algorithm, we remove the congestion from the sockets allowing the Workers to send replies to Coordinator more quickly, with data of greatest accuracy. I conclude that my hypothesis about performance problem is correct!

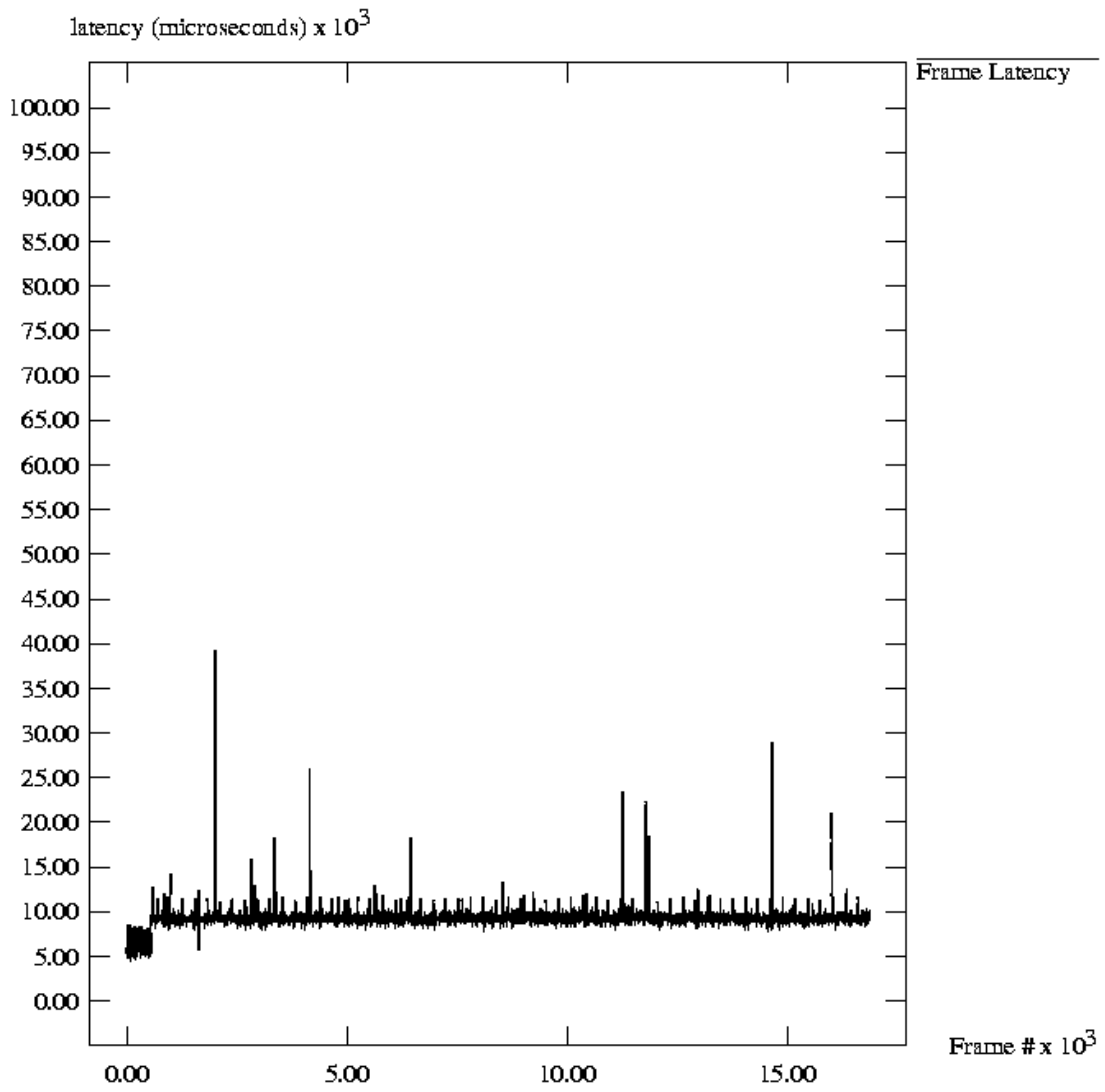
The following graphs show the accuracy of the frames using the modified algorithm:



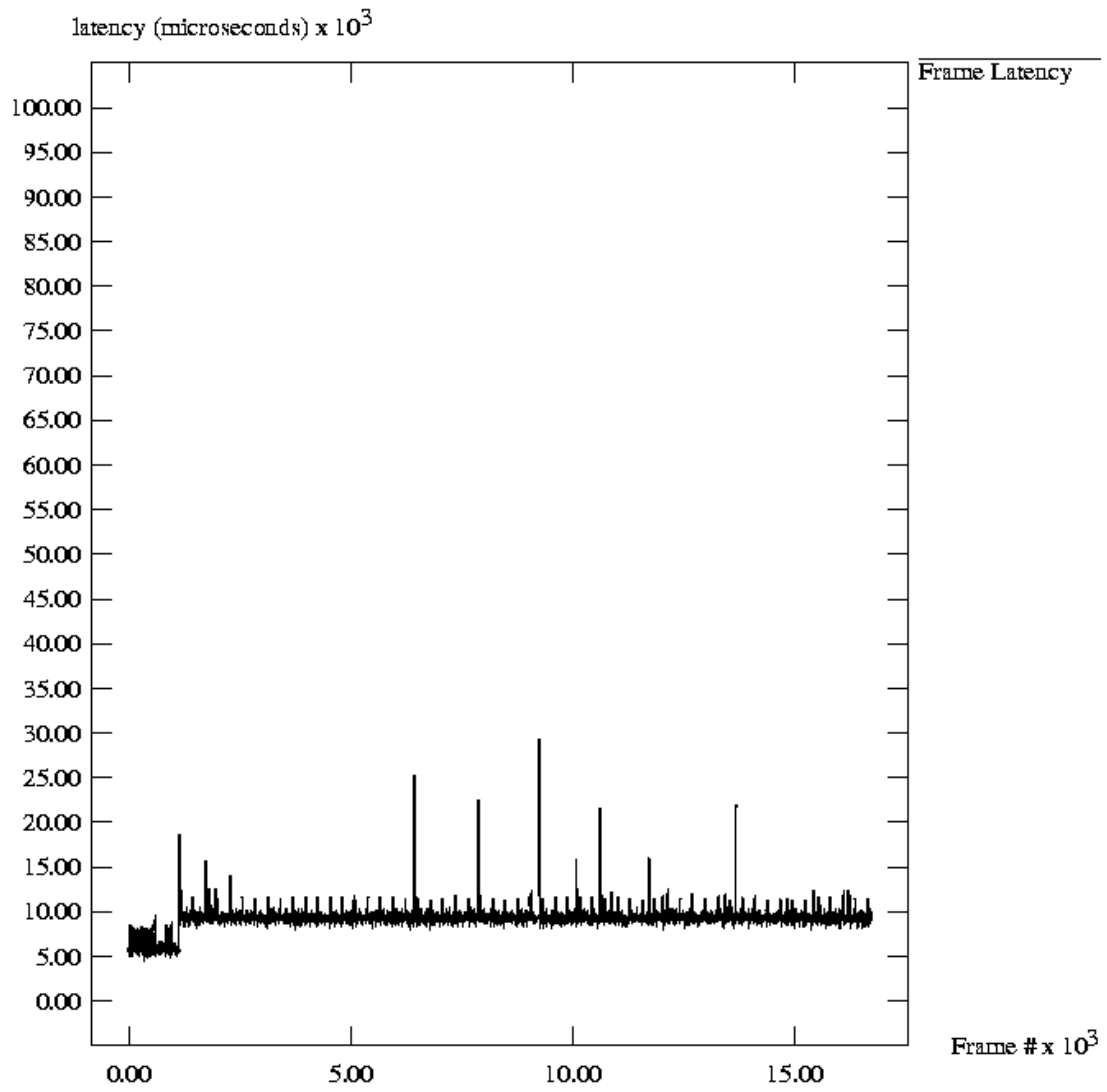
GraphID(101F) drop=30, APE=100



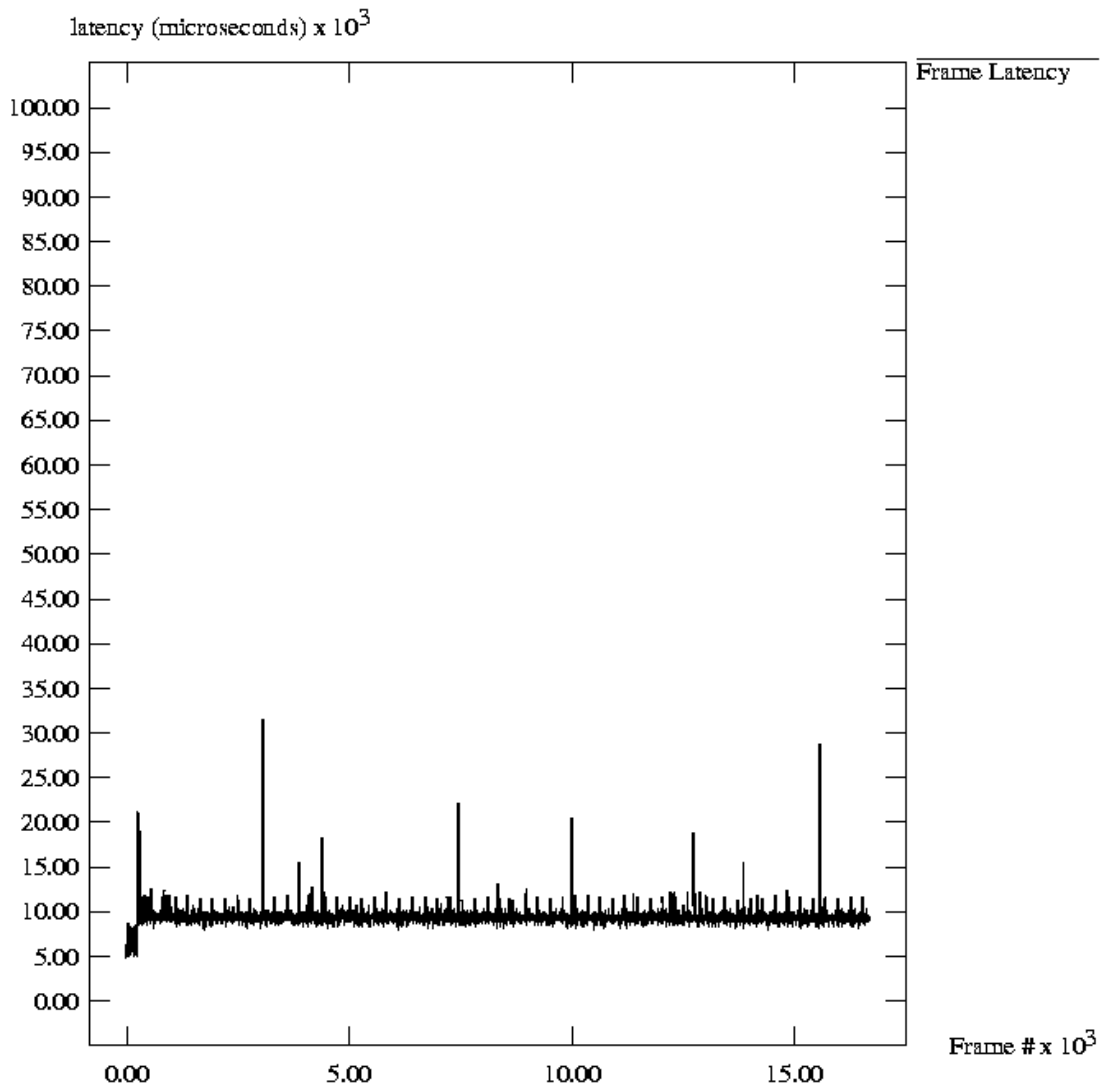
GraphID(102F) drop=300, APE=100



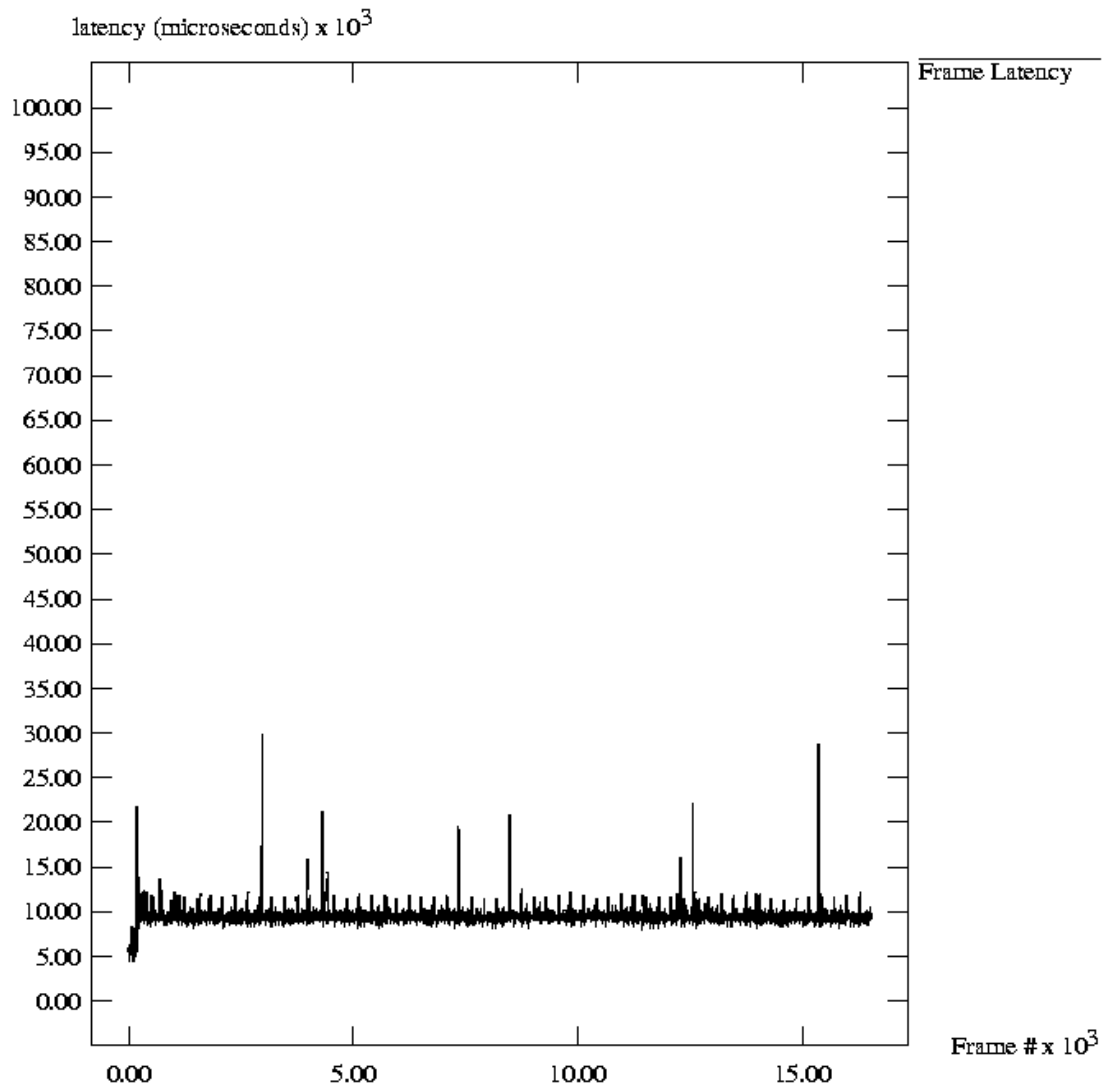
GraphID(103F) drop=3000, APE=100



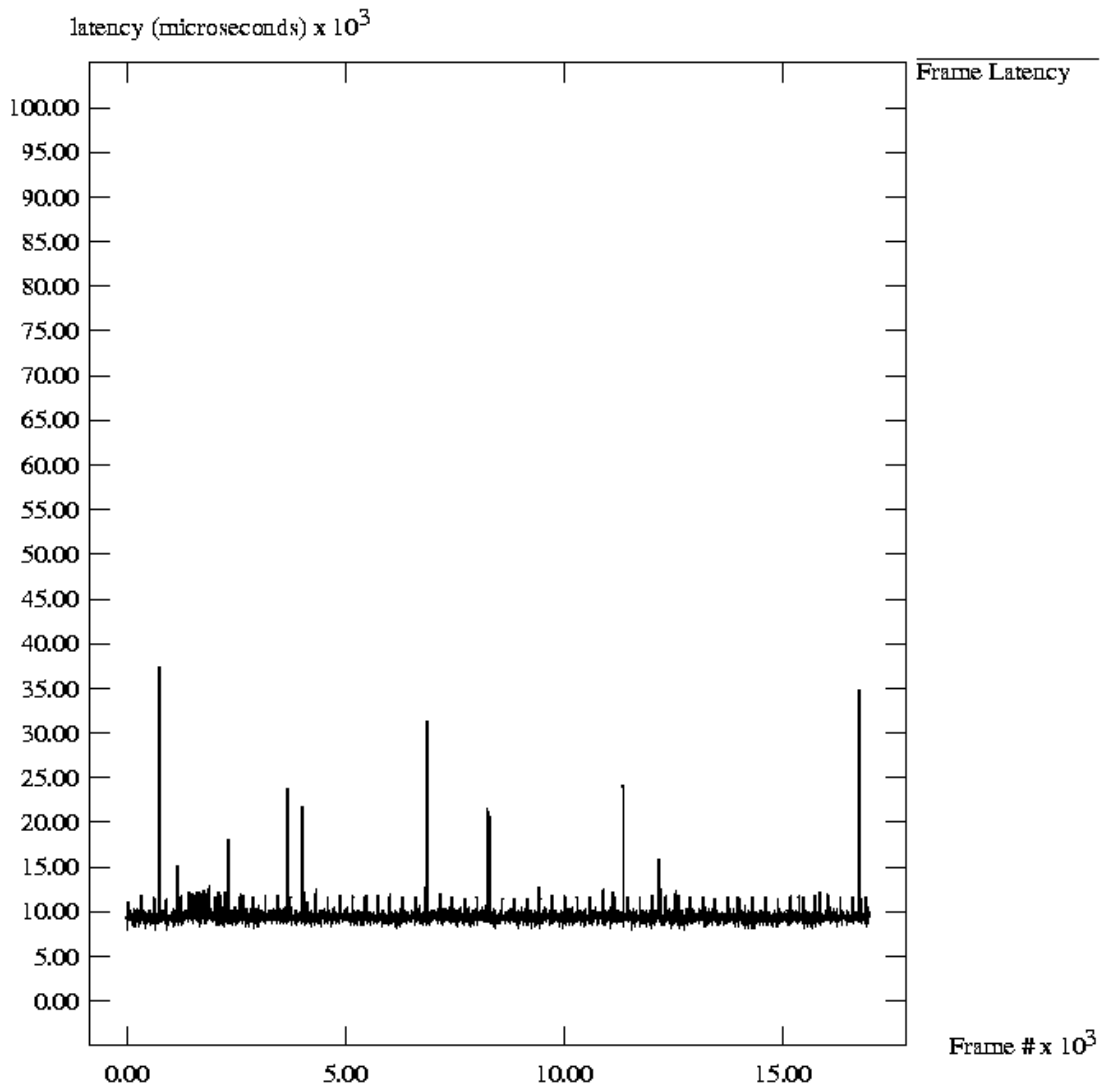
GraphID(104F) drop=30, APE=1000



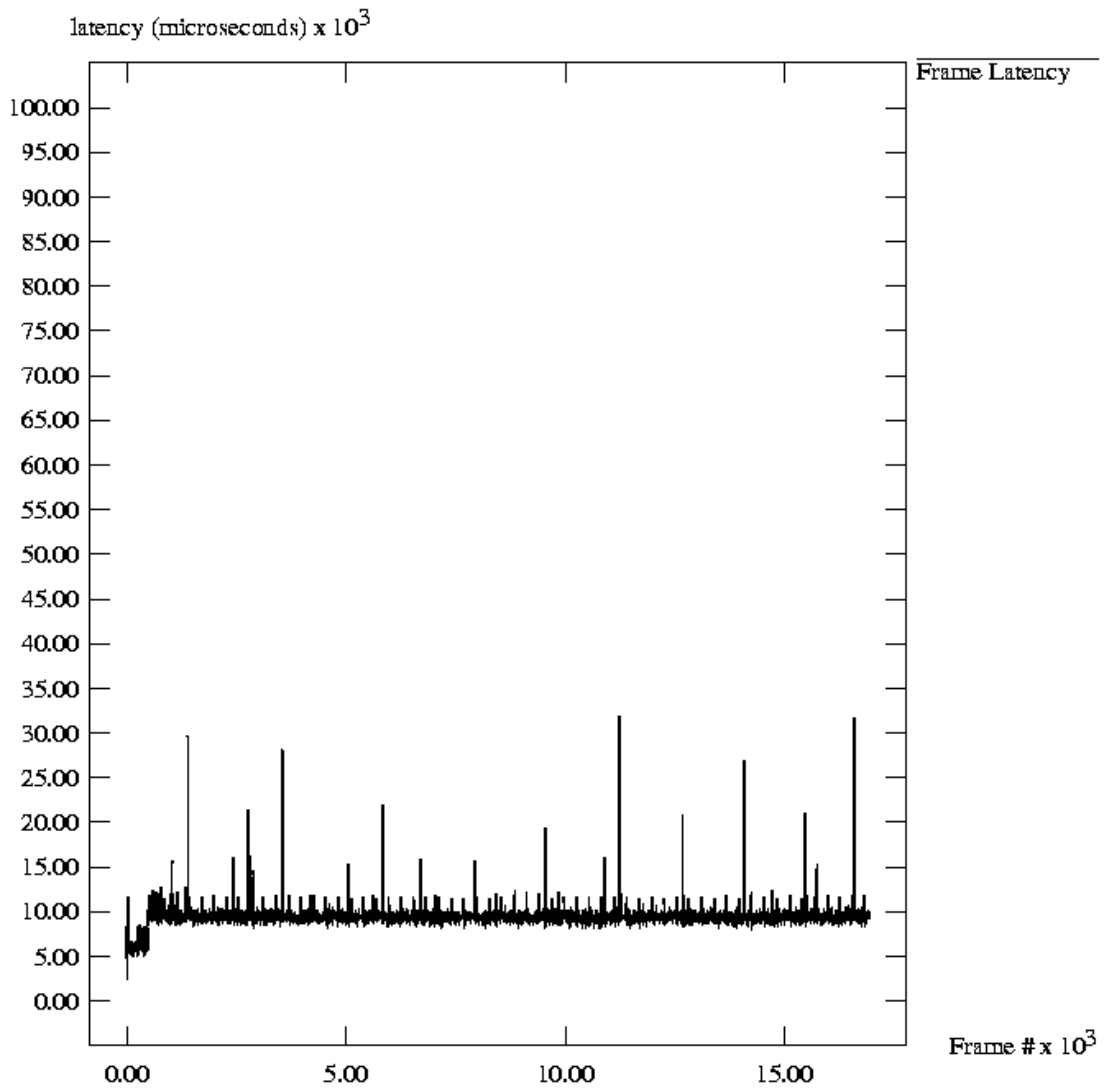
GraphID(105F) drop=300, APE=1000



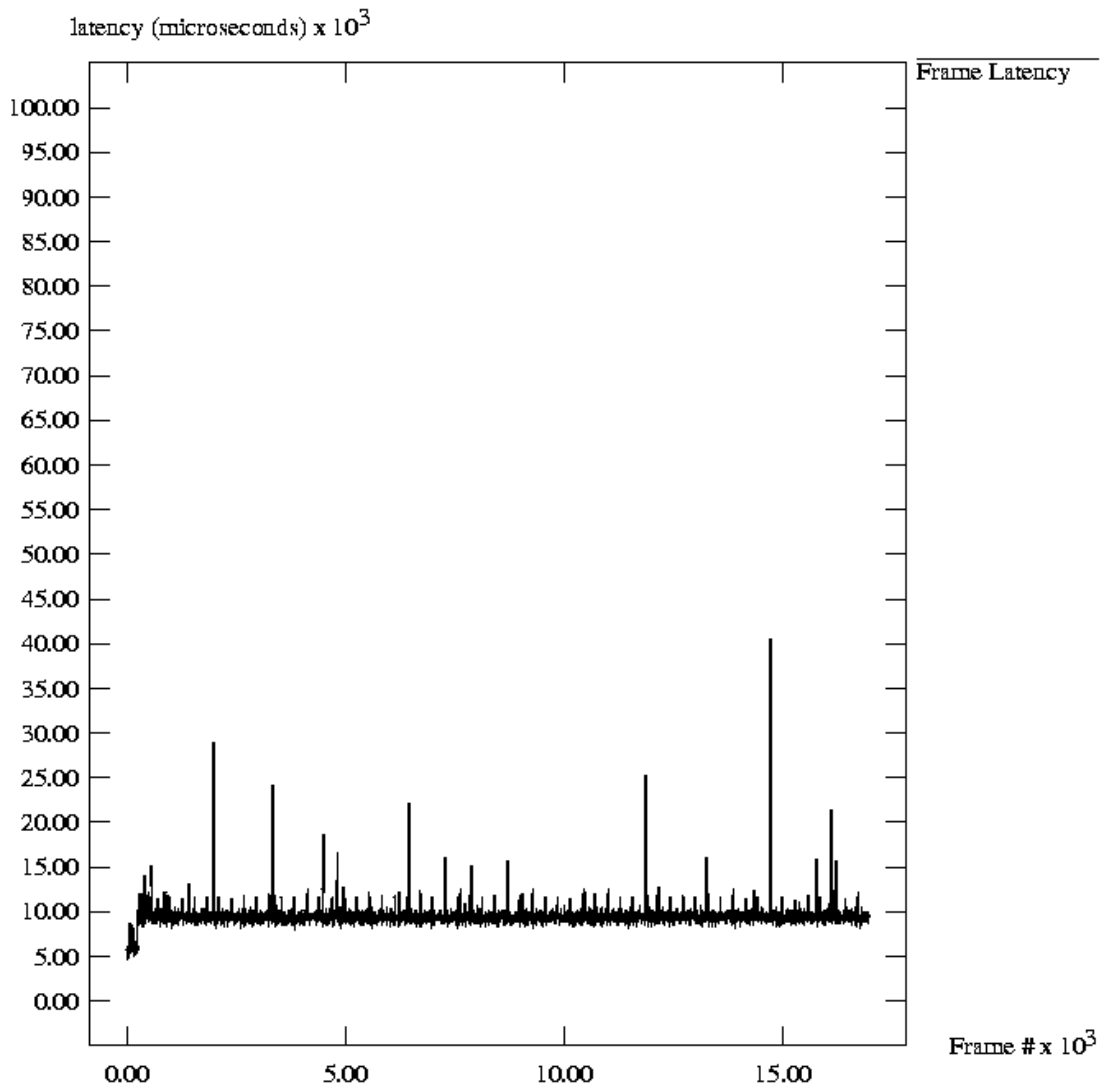
GraphID(106F) drop=3000, APE=1000



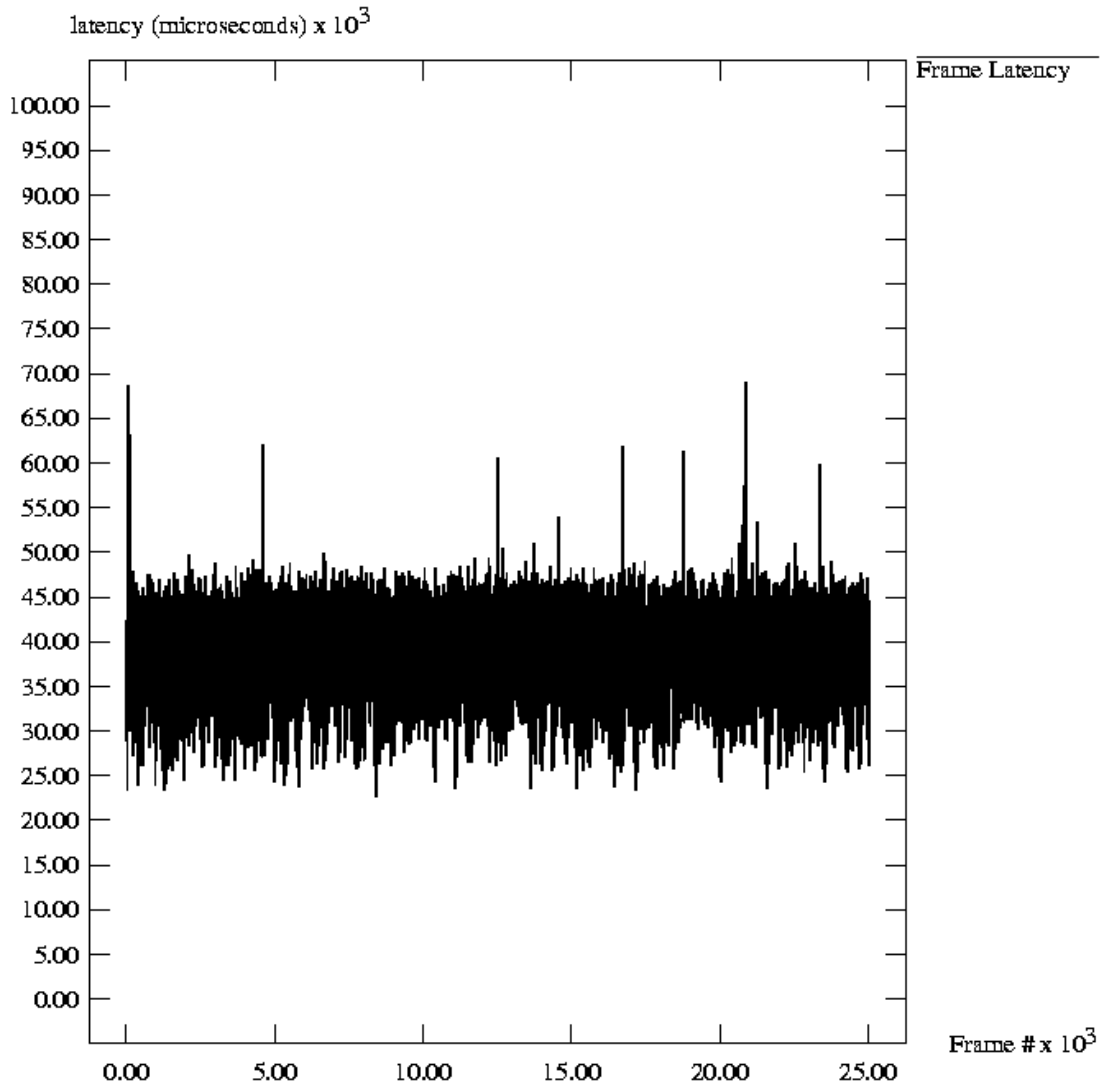
GraphID(107F) drop=30, APE=10000



GraphID(108F) drop=300, APE=10000

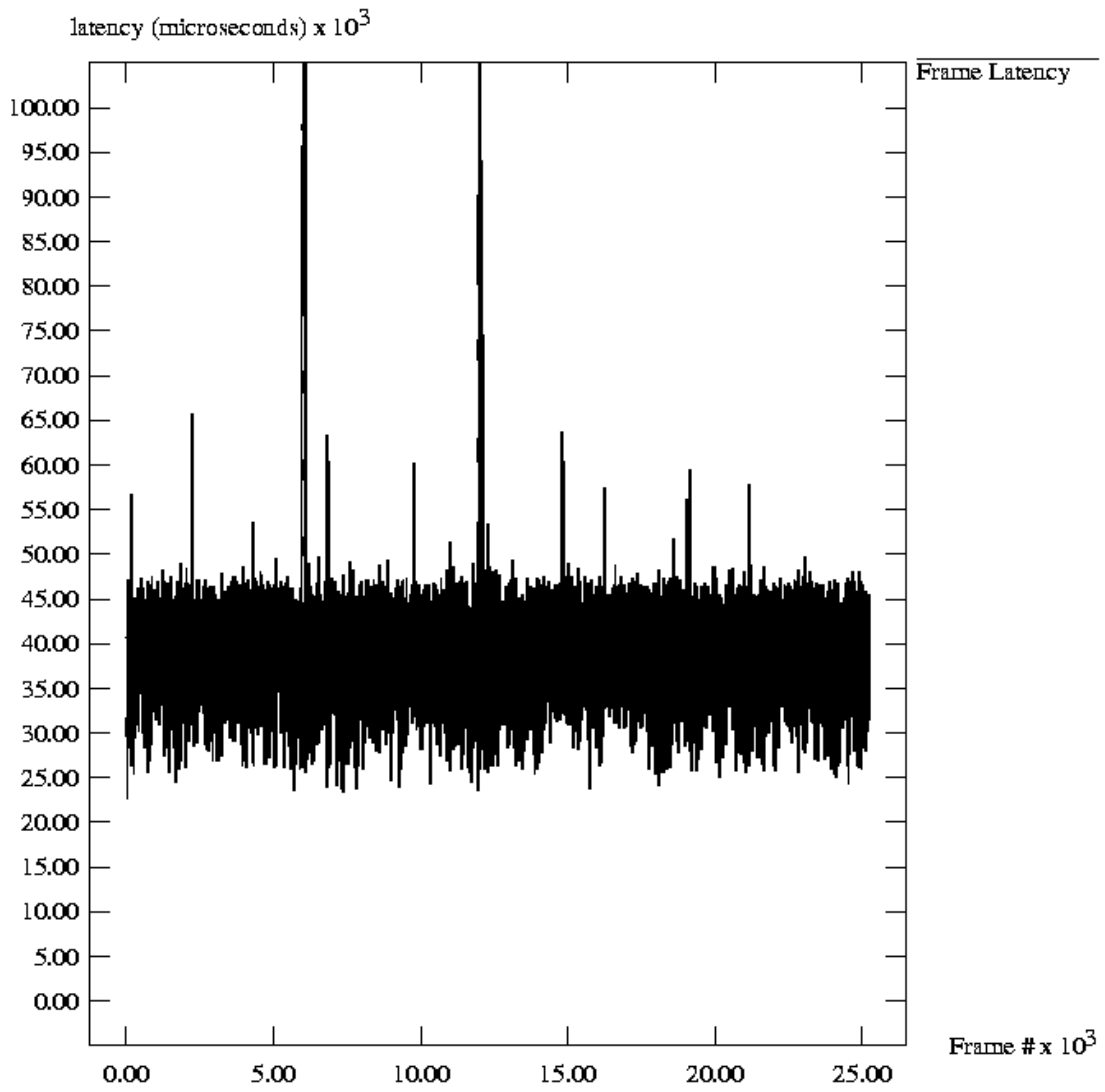


GraphID(109F) drop=3000, APE=10000

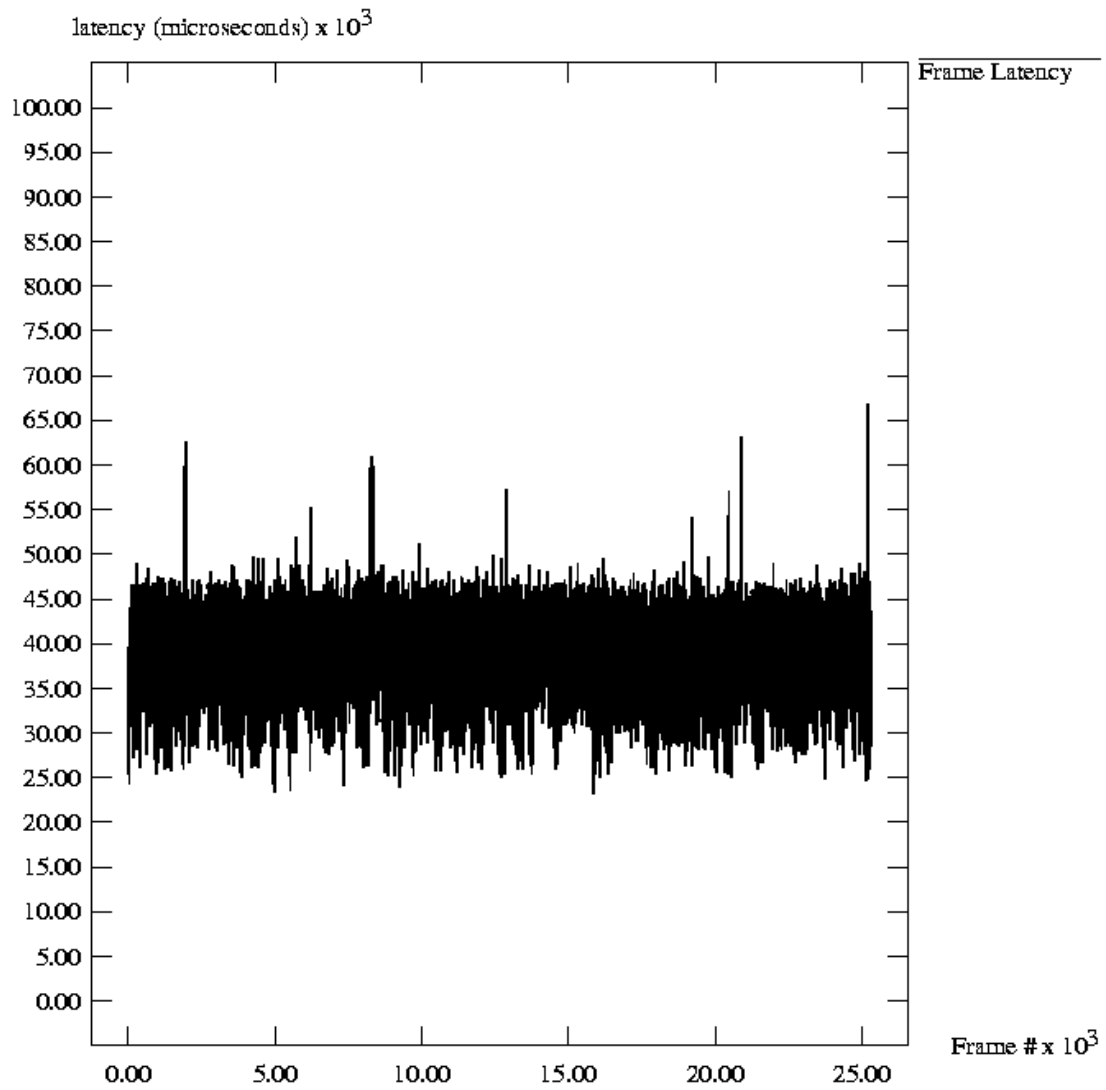


GraphID(110F) drop=30, APE=100000

The above and the following two graphs show that frame accuracy varies between 25 and 48 msec. This is due to the high value of the APE variable. The statistical skew uses the difference between the oldest request of all output Variables and the wall clock. Since these requests are computationally expensive (APE = 100000) they will come back to the Coordinator as replies with great latency.



GraphID(111F) drop=300, APE=100000

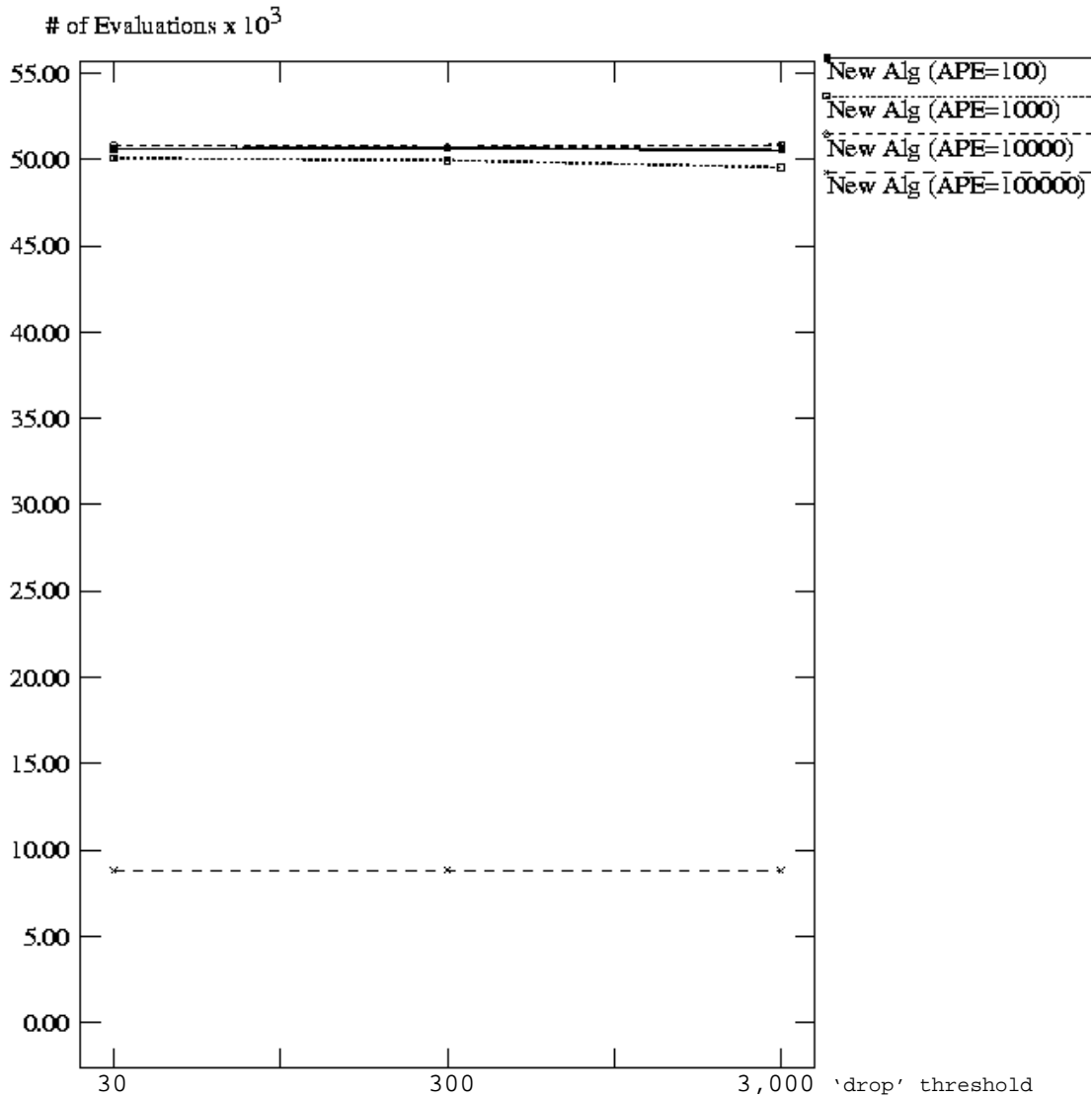


GraphID(112F) drop=3000, APE=100000

Throughput in DLoVe

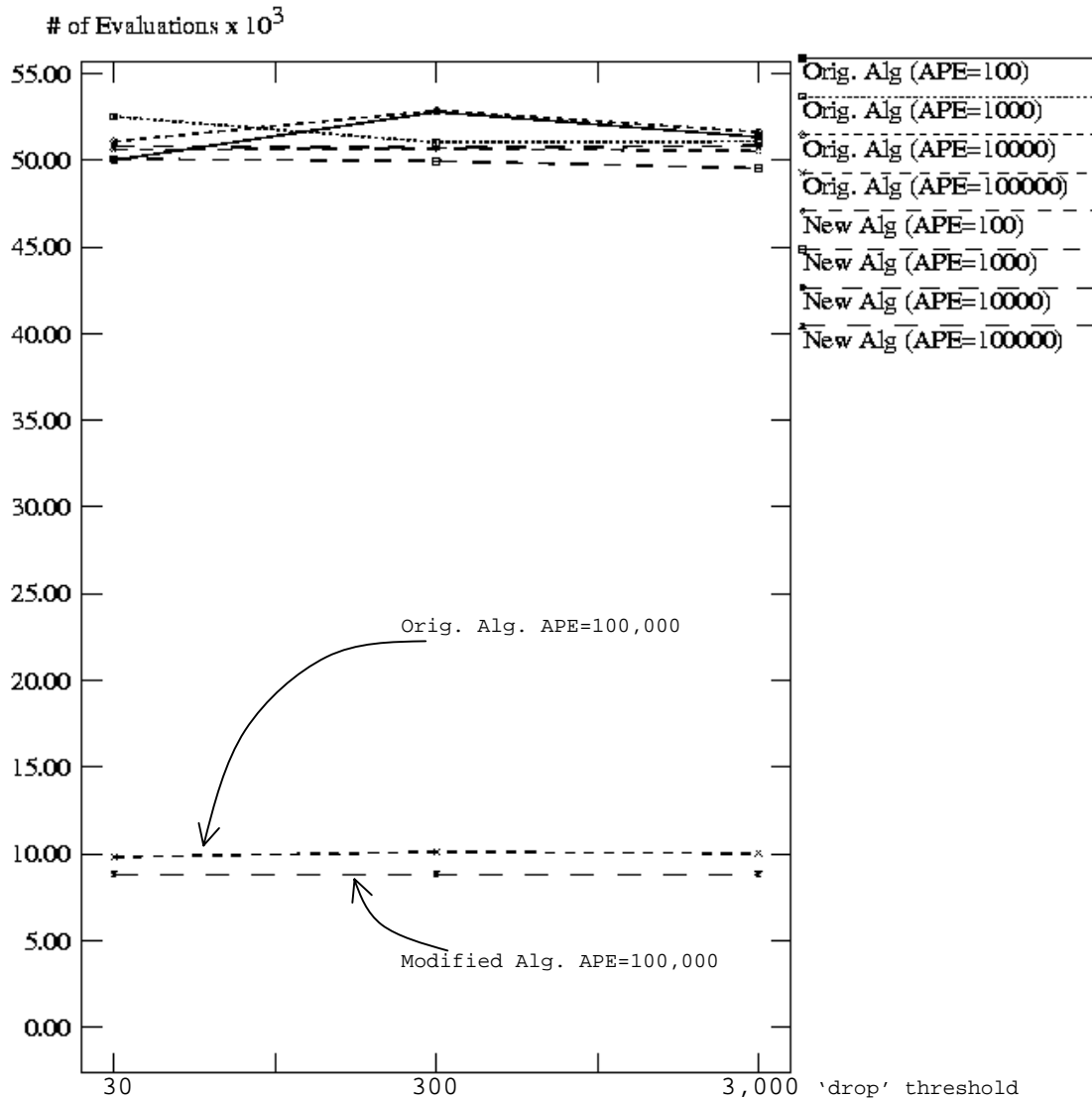
The modified algorithm of the Workers not only fixed the stepwise increase in the message latency, but also improved dramatically the accuracy of the frames with very little throughput penalty.

The following graph shows the number of evaluations achieved by the modified algorithm of the Workers.



Throughput of DLoVe utilizing the new algorithm on the Workers

The following graph shows the number of evaluations achieved by both the original and the modified algorithm.



Comparing throughput between the original and the new algorithm

The penalty in throughput using the modified algorithm is only 12% in return for much more accurate frames. Also the number of frames should be observed that increases dramatically when evaluations are computationally expensive (APE=100,000). Comparing the graphs with GraphID 010, 011, 012 against 110, 111, and 112 the increase in frame rate can be visualized. Using the old algorithm

the Coordinator achieves 56 frames/second (3400/60) where using the modified algorithm the Coordinator achieves 416 frames/second (25000/60). Note that the 'Perf' program does not render the display. Throughout these experiments, I assume that when a frame is rendered it takes zero time.

The number of evaluations in the non-distributed version is not plotted due to its wide range of numbers, where plotting the data would not be very useful. The following table shows that when evaluations are computationally inexpensive (APE < 10,000), which means that it is better performing the calculations locally than sending requests over the network the non-distributed version outperforms the distributed version. However, when the calculations are computationally expensive, the distributed version outperforms the non-distributed version by a factor of 2.9 (APE = 100,000).

drop threshold	30	300	3,000	30	300	3,000
non-distributed	582,008	582,008	582,008	201,686	201,686	201,686
dist. New	50,606	50,677	50,535	50,070	49,962	49,542
APE	100			1,000		

drop threshold	30	300	3,000	30	300	3,000
non-distributed	26,066	26,066	26,066	3,036	3,036	3,036
dist. New	50,827	50,721	50,833	8,809	8,813	8,816
APE	10,000			100,000		

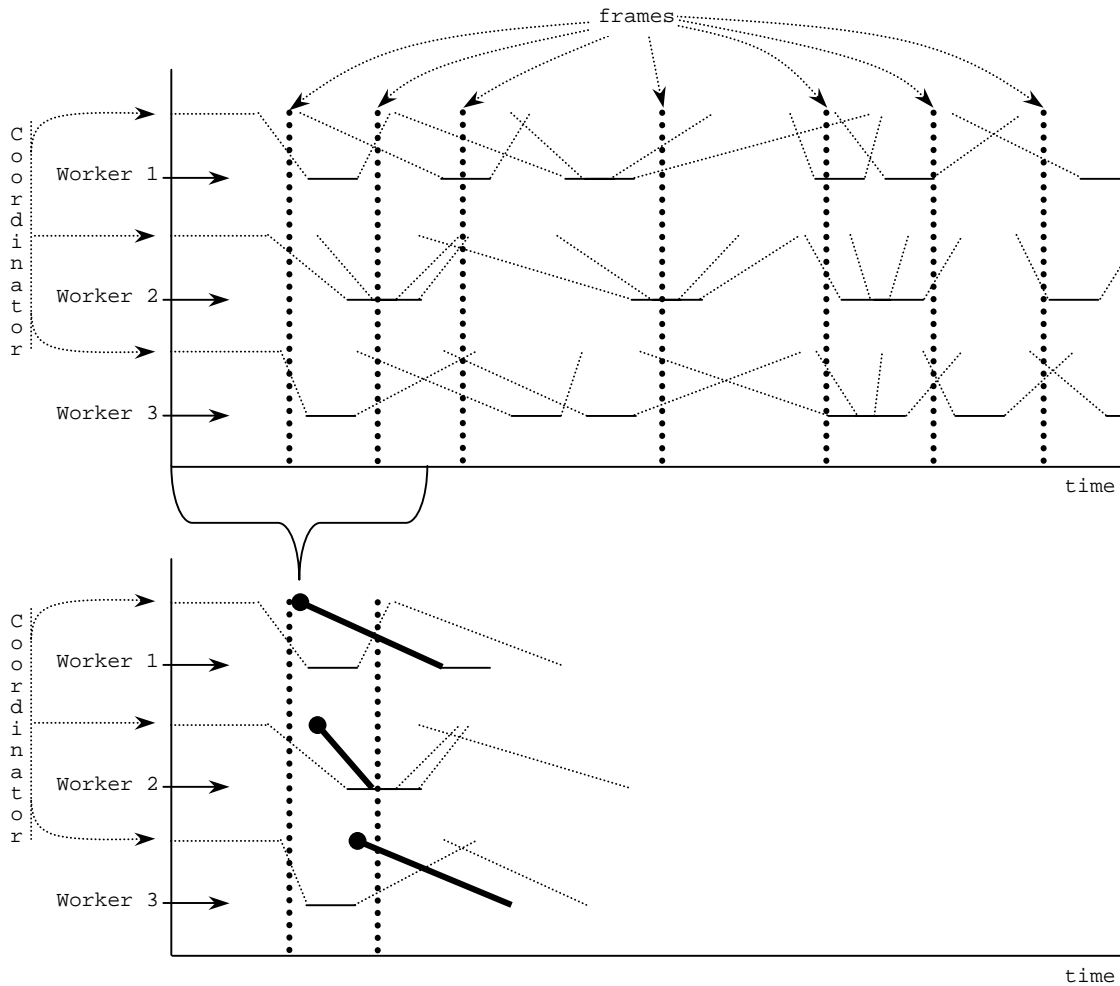
Comparing throughput between non-distributed and distributed version of 'Perf'

When calculations are computationally expensive, using the modified version of the algorithm we achieve more frames per second; that enables the Coordinator update the display more frequently. Updating the display more frequently does not yield anything additional if the data is old or the same as the 10 previous frames, because messages did not come back from the Workers. However, the camera and hand

position and orientation in DLoVe are calculated locally since these are critical factors in any virtual environment. Even though the simulated objects may be displayed the same for the past 10 frames for example, navigation and immersion of the user in the virtual environment is achieved because of the larger number of frames per second.

Asynchronous Frame Rate in DLoVe

It does not make sense to measure frame rate in DLoVe because the frame rate is asynchronous and even if we measure it, we cannot tell how fresh the data is that the Coordinator uses to render the display. The following graph shows why the frame rate in DLoVe is asynchronous:



Asynchronous rendering in DLoVe

The Coordinator renders the display as soon as it sends the data to Workers. If the network is congested, the Coordinator spends more time trying to send the requests to Workers. Also, depending on how loaded the network is the propagation time from the Coordinator to Workers and visa versa varies. As a result, the frame rate is not constant because it depends on the time the Coordinator spends sending requests to Workers.

Summary of Results

The experiments demonstrated that DLoVe not only describes specification of Virtual Reality programs well, but also improves overall performance of applications designed in its framework by dramatically increasing the validity of the rendered frames. In addition, DLoVe supports mechanics for implementing or transforming single user programs into multi-user programs.

Chapters 10, 11, 12 demonstrated that DLoVe can be used to implement large scale Virtual Reality applications, and when speed is required, DLoVe's framework is able to provide the additional CPU cycles needed by real time application by utilizing multiple machines. The modified algorithm on the Workers proved to be successful in eliminating the network congestion and providing more valid frames with a small throughput penalty of about 12 percent.

This chapter showed the need for a different method of measuring performance that describes DLoVe accurately, where traditional methods fail by providing seemingly acceptable performance measures that in actuality reflected poor performance. Throughput is not enough to describe performance, but throughput in conjunction with statistical skew do accurately describe DLoVe's performance.

More experiments need to be conducted to understand how DLoVe behaves in high-speed networks and how multiple Coordinators influence the performance of DLoVe. However, because appropriate equipment was unavailable, these experiments must be deferred for future investigation as described in the next chapter.

Chapter 14: Evaluation

Overview

Chapter 13 measures and evaluates the distributed and parallel aspect of DLoVe. It discusses problems and resolutions of measuring the distributed and parallel version of DLoVe. Chapter 13 demonstrates that DLoVe performs well by measuring both overall throughput and frame latency. This chapter evaluates DLoVe by looking at it primarily as a paradigm for designing VR applications for a single user and a single machine.

DLoVe is designed to provide a framework to programmers for defining and implementing both serial and distributed VR applications for single or multiple users. Its architecture includes features and techniques that explicitly address issues encountered when developing applications such as virtual environments and multi-user VR interfaces. Test applications were developed and implemented using these features, and were discussed in chapters ten, eleven, and twelve.

Conceptual Applicability

While in the real world processes exhibit continuous changes in state, any computer model of these changes consists instead of a discrete change of measurement events. This, however, should not influence the programmer's understanding of how these events interrelate. It is this conceptual understanding that should be captured in the design, not the mechanics of its implementation.

DLoVe supports specification mechanisms that allow interaction object behavior and manipulation to be described in both continuous and discrete terms. It supports mechanics to allow the designer to combine continuous and discrete domains to express his/her conceptual model being designed. The responsibility of running a program in distributed mode falls upon DLoVe itself.

Scalability Issues

Virtual Park demonstrates that DLoVe is capable of creating large-scale applications employing non-WIMP interactions. The park demonstrates that DLoVe's programming paradigm results in a specification that is relatively quick and easy to implement. The specification modules themselves e.g. ball, arm1, arm2, etc, retain

the ability to perform low-level manipulation where needed, and combined produce a higher level of abstraction in the program without higher levels of complexity.

Even though the number of Workers (when a DLoVe application is executed in a distributed environment) can be increased by modifying a single line in the configuration file, the number of Workers that can practically participate is limited. The bottleneck preventing parallelism is the bandwidth of the physical network connecting the Workers and the Coordinators.

DLoVe applications would execute much faster in a shared memory machine. But even in this case the number of processes could not exceed twenty, because of the lack of scalability in the shared memory environment itself. Even if twenty Workers run on a single shared machine, networking would still be necessary when designing multi-user applications. Applications designed using DLoVe's paradigm exhibit larger speedup when evaluations are computationally expensive, but the distributed version offers no benefit, when evaluations are inexpensive. However, even in the case where DLoVe does not exhibit any speedup, due to lack of constraint complexity, it still supports multi-user application development.

Modifying a single line in the configuration file allows one to increase the number of Coordinators participating in a distributed environment. However, one must also modify code to inform all the modules of new input and output devices.

Extensibility Issues

DLoVe is designed to adapt to the evolving needs of non-WIMP user interfaces. To remain viable DLoVe must handle new input and output devices and the introduction of new interaction objects. Encapsulated Links and Variables within device drivers allow external devices to be introduced to the system without requiring any modification to the UIDL. For example, DLoVe does not care if the user reads from the Polhemus 3D tracker, an eye tracker device, or inputs from an X window. The mechanics of reading from any of these devices are hidden from the designer. The sample applications presented in Chapters 10, 11, and 12 include an interface to X Windows, an interface to the Performer graphics engine, a device driver for Polhemus 3D tracker, and in chapter 11 a device driver for the eye tracker.

In a multi-user environment, different users can utilize different input/output devices to interact with each other and with the same virtual world. When the 'arms' program (chapter 10), the eye tracking program (chapter 11), and the Virtual Park (chapter 12) are executed in a two-user environment, one user operates the head and hand position, and orientation with the mouse, while the other uses the Polhemus 3D tracker. One user uses the monitor as an output device and the other an HMD.

Preserving Intellectual Investments

New tools, programming languages, and any kind of software, that takes longer to apply to any given problem than to solve the problem manually will eventually be abandoned by users. In an effort to avoid this, DLoVe seeks to capitalize on existing notations, concepts, and standards wherever the opportunity presents itself. Feedback from students using DLoVe for their programming assignments suggests that embedding new concepts within familiar programming methods provides a stable launching point for exploring and experimenting with the new paradigm. The ability to call C++ procedures protects the substantial investment of academia and industry in this language.

Chapter 15: Open Problems and Further Study

Overview

DLoVe has proven that it is an applicable paradigm for designing and implementing non-distributed and distributed Virtual Reality applications. It performs relatively well in a networked environment. It supports mechanics for transforming serial programs into distributed programs by following a simple pattern. It also supports mechanics needed for multi-user applications. Several open problems with DLoVe should be addressed. DLoVe's paradigm can also be enhanced by further work.

Enhancing and Improving DLoVe's Paradigm

DLoVe assumes that all objects in a virtual environment are created before the application begins reading input devices. If the Coordinator creates any new objects after the `Link::InitSystem()` call, DLoVe's protocol cannot notify the Workers of those changes. When a DLoVe application is executed in non-distributed mode, however, there is a workaround allowing dynamic creation of objects. DLoVe's protocol must be extended in order to allow Workers to adapt to dynamic changes. This is needed, for example, when a user selects car parts to design a futuristic car model. Each created part becomes a new object in the constraint space.

Many enhancements are possible in how DLoVe automatically handles code changes, needed to support multi-user interfaces. To change a single-user interface to multi-user, all input Variables (and the Links to which these input Variables are attached) need to be duplicated and the network re-wired – for the duplications. Since this process is well understood, it is possible to automate it and remove this burden from the programmer. Chapters 10 and 12 describe, using examples, the process of changing a single-user interface into a multi-user interface.

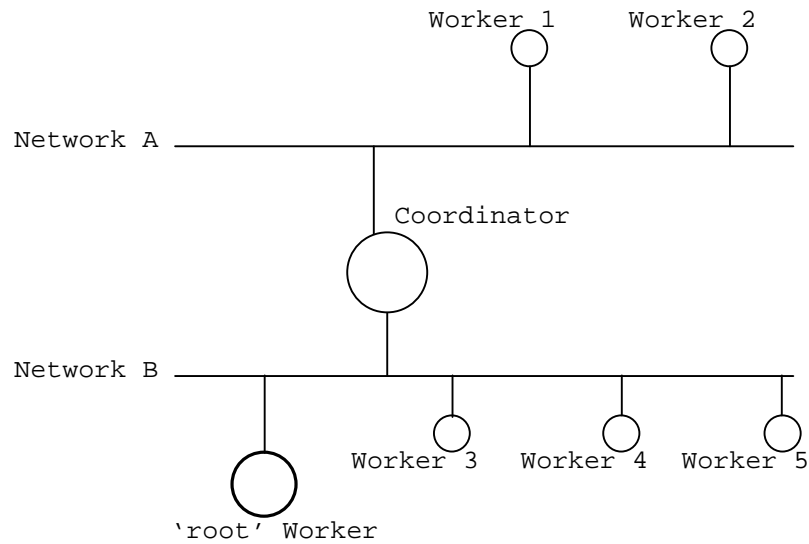
In DLoVe's current paradigm, the programmer must manually introduce artificial auxiliary Variables to combine multiple output Variables from a single Link so that those output Variables will not automatically be assigned to different Workers, thus causing redundant computations. This situation is easily detectable automatically by BFT of the constraint graph, so we could automate this process and save manual effort. An illustration of artificial auxiliary Variables is given in the Virtual Part application where each Humanoid is simulated entirely by a Worker.

DLoVe's partition algorithm partitions the constraint graph assuming that all Links take comparable computation time, so that DLoVe's task assignment does not balance loads for structures for which Links vary greatly in expense. To achieve load balancing, the programmer must assume that all Links are roughly equivalent in cost and that Workers are equal in performance. To alleviate this responsibility, we could improve DLoVe's partition algorithm to account for task expense and machine speed. Currently, partitions are determined at compile time but should be determined dynamically based upon the structure of the graph and machine performance. It is unclear whether dynamic load balancing would improve many programs, due to the overhead involved in rebalancing.

In the current implementation of DLoVe's protocol, programmers have to deal with cyclic graphs by cutting the constraint graph vertically and introducing synthetic Variables. Without synthetic Variables, programs with cyclic dependencies cannot be executed in a multi-Worker environment, as illustrated in Chapter 12. Synthetic Variables are used for inter-dependencies among simulated objects such as the Humanoids in the Virtual Park. DLoVe's partition algorithm could be extended to automatically insert synthetic Variables where needed, by using a BFT algorithm, and also to insert code that will update all the synthetic Variables.

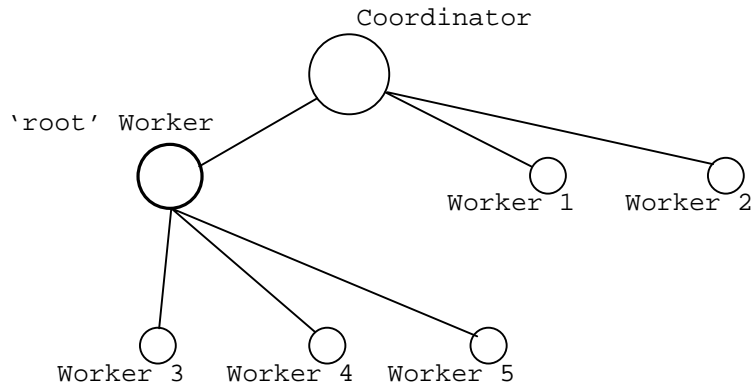
Hierarchies of Workers could increase the performance of DLoVe when synthetic Variables are introduced in a program. DLoVe's partition algorithm partitions the constraint graph by cutting it horizontally. Use of synthetic Variables requires a vertical cut of a cyclic constraint graph in order to execute a program in parallel. Sub-graphs of the constraint graph that contain synthetic Variables can be assigned to a special set of Workers to handle updates in parallel. There can be a 'root'

Worker that will use other regular Workers to update synthetic Variables. The 'root' Worker with its Workers can exist on a different network to avoid network congestion. The following diagram describes this environment:



'root' Worker incorporated in DLoVe to handle synthetic Variables in cyclic constraint graphs

Workers 3, 4, and 5 are connected to the 'root' Worker that communicates Variable updates to Coordinator. These machines communicate with each other as shown below:



Node connection with the 'root' Worker

The 'root' Worker can be a similar process to a regular Worker. The difference is that the 'root' Worker synchronizes data delivery of synthetic Variables to Coordinator. This separates the network traffic into two types of messages. Request messages directly from the Coordinator, and synthetic Variable updates.

My research suggests that by re-implementing DLoVe using UDP instead of TCP, its performance will greatly be increased [Singhal 99]. TCP is a reliable protocol but it does not best characterize DLoVe's transactions. If some requests, such as SetE and GetE, are lost, they do not impact programs using DLoVe's paradigm due to the frequency of requests. In cases of Enable and Disable requests (that re-wire the constraint graph), requests can be initiated with an acknowledgment requirement. The Coordinator will keep a vector of which Workers need to be updated on such requests and the acknowledgments it received from the Workers. If such messages are lost, the Coordinator will retransmit the state of the constraint graph. This technique will improve the performance of DLoVe due to using a less reliable but significantly faster protocol (UDP) than TCP.

My research suggests that multicasting may be more efficient in distributed environments as an alternative to unicasting over TCP or UDP. Currently, the Coordinator must send each message to each Worker individually. This means that communication is proportional to the number of Workers. Multicasting would enable the Coordinator to deliver a message to multiple Workers with a single send operation. However, the Coordinator still must poll each Worker individually for any replies they might have. But this will decrease the time needed to send a request out to the Workers.

Failures on Workers can be detected by the Coordinators, which can then redirect network traffic to still functioning machines. All systems in DLoVe's distributed environment have an exact copy of the constraint graph. All machines are capable of bringing Variables up to date and replying to Coordinators' requests. In the current implementation when a failure is detected all processes terminate. DLoVe's protocol can be very easily extended and automatically redirect traffic to still functioning machines when failures occur. This may require the Coordinators to re-execute the partition algorithm to achieve the best performance possible.

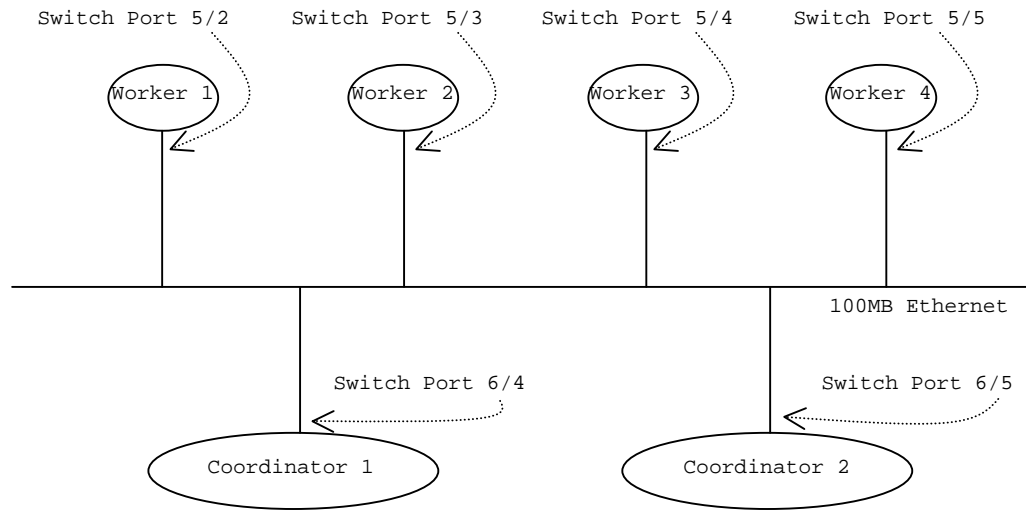
In a multi-user environment, a crash or a disconnect of one Coordinator should not result in the termination of the entire simulation. The current implementation terminates the simulation if any of the Workers or the Coordinators terminate. DLoVe's protocol should be more flexible to allow Coordinators to connect and disconnect while the simulation is executing.

To make DLoVe a software package that can be used even more easily by programmers, it must also contain a GUI interface where designers and programmers can define Links, Variables, EventHandlers, and their relationships, in

a visual fashion. This GUI should also be able to generate C++ code, or read C++ code and display the visual representation of the constraint graph.

To understand even more how the type and speed of a network, and the speed of the machines (such as Sun ultra 250 servers) influence the performance of DLoVe more experiments need to be conducted on different types of networks such as a 100MB or Gigabit switched Ethernet. The state-of-the-art Ethernet is the Gigabit Ethernet with two major advantages over regular Ethernet. First, it preserves Ethernet's simplicity while enabling a smooth migration to Gigabit-per-second (Gbps) speeds. Second, it delivers a very high bandwidth to aggregate multiple Fast Ethernet segments to support high-speed server connections, switched backbones, and high-speed workgroup networks [Buyya 99a].

Moreover, more experiments need to be conducted to understand how much multiple Coordinators influence the performance of DLoVe. Multiple Coordinators imply increase in the number of requests sent to Workers, thus increasing workload on the Workers as well as on the network. The following figure shows a the connections of multiple Coordinators on a high speed switched network, such as 100MB or Gigabit switched Ethernet:



Performance Measurement of a 100MB switched network using multiple Coordinators

Chapter 16: Conclusion

DLoVe was designed to provide a specification paradigm and a framework to assist in defining and implementing non-WIMP interfaces. It allows programs to be executed in a distributed or non-distributed environment where speed is a requirement without major code modifications. It allows programmers to design and implement VR applications in a high level of abstraction where their code can be modular and re-useable. It allows easy specification of functionality for multi-user interfaces, following a simple pattern. Its run-time engine is responsible for performance optimization and network control; problems faced by distributed non-WIMP user interfaces. It hides all the networking aspects of message passing among the machines participating in a distributed environment. As a result, the DLoVe programmer does not need to understand distributed and parallel systems to employ DLoVe. The programmer need only be familiar with C++ one of the most common languages for application development.

DLoVe was tested against each of these claims and found to be successful and robust. Utilities are provided by DLoVe to allow fine performance tuning of applications designed within its framework. My own as well as others' experiments

of solving problems and implementing applications suggest that DLoVe is successful in achieving its goals and meeting the needs of non-WIMP user interface designers.

DLoVe is a new specification paradigm for designing and implementing VR applications. Applications designed to be executed in a single machine can also be executed in a distributed environment involving multiple workstations, with minor code modifications to achieve greater throughput.

Chapter 13 demonstrates how well DLoVe performs when calculations are computationally expensive. Chapter 13 also shows the need for a different method of measuring performance that describes VR systems accurately, where traditional methods fail by providing seemingly acceptable performance measures that in actuality reflect poor performance. Throughput is not enough to describe performance, but throughput in conjunction with statistical skew do accurately describe DLoVe's, and in general VR systems', performance. Multi-user applications can also be designed or translated from single-user into multi-user with little programming effort following a simple pattern. Chapter 10 illustrates how a program designed for a single user can be translated into a multi-user one.

Appendix A

The directory hierarchy of the entire DLoVe software is shown below. The top level directory of the hierarchy is the directory 'implem' that contains several subdirectories that as shown below:

- basic
- data
- opt
- solve
- eye
- net
- scripts
- state
- use.x
- use.pf
- use.eye
- use.perf
- use.pf.Humanoids
- use.pf.arms
- use.pf.2Zoom

Directory: basic

In this directory reside the files that describe the part of the UIMS. It contains code for reading input devices and uses it to operate plugboard Variables and events.

Directory: data

Any 3D models the user wants to use with his program are going in this directory.

Directory: opt

This directory contains optional utility features that are built on top of the files in 'basic'. These are utilities that a programmer can use, or build equivalent ones to fit new environments. The following modules are included in the 'opt' directory:

- Links
A collection of pre-defined Links for some common arithmetic operations.
- Quitter

A simple EventHandler object that accepts an ESCAPE key on the keyboard and exits the program.

- **PfWindow**

This is a convenience class for use with the Performer graphics engine. It initializes Performer, creates the window, and creates the corresponding DeviceXWindow, which is used for reading input.

- **XWindow**

This is a convenience class for use with X (not Performer). It initializes X, creates one window, and creates the corresponding DeviceXWindow.

Directory: solve

This directory contains DLoVe's source code of the constraint solver. It also contains the code with which the programmer can operate on the Variables such as SetE(), SetI(), GetI(), and GetE() functions.

New constraint solvers can be written by replacing these files with code of a new constraint solver. Currently, besides DLoVe's constraint solver, there is another simple-minded and inefficient constraint solver, which was designed about 4 years ago (this was by Prof. Jacob, and it was the starting point of this dissertation). New constraint solvers can be written simply as subclasses of VariableBase and LinkBase, modules residing in 'basic' directory.

Directory: eye

The 'eye' directory contains code for reading and processing data received on serial port from ISCAN eye tracker (kept in separate directory, because most sites will not have this device). The Polhemus data is read via the eye tracker and not directly from Polhemus.

Directory: net

This directory contains the code needed in making distributed and multi-user interfaces. All necessary network programming is contained in the files in this directory and no additional user network programming and learning is required.

Directory: scripts

Scripts for performance analysis reside in this directory. These scripts manipulate data coming directly out of DLoVe to measure frame validity, message latency, and other things.

Directory: state

This state diagram translator resides in this directory, which is part of an earlier system for processing and executing state diagrams (designed by Prof. Jacob). This state diagram translator has many features not required for the present usage. If for any reason dealing with the state diagram translator proves troublesome, one can still write his/her `lho()` routines in plain C++ with "if" statements rather than using the state diagram notation, and remove the "translate5" lines from the makefiles.

Multiple directories: use.*/*

These directories all contain code that demonstrates the use of the UIMS, including Links and Variables, defined for each particular user interface one wants to build.

- `use.x`

This directory contains example and skeleton programs without Performer (using X or using no graphics)

- `use.pf`

This directory contains example programs using Performer graphics.

- `use.eye`

This directory contains example programs using the eye tracker and Performer.

- `use.perf`

This directory contains the code for the 'Perf' program that was used for the performance analysis of DLoVe (chapter 13).

- `use.pf.Humanoids`

This directory contains the code for DLoVe's Virtual Park application described in chapter 12.

- `use.pf.arms`

This directory contains the code of the 'arms' application described in chapter 10.

- `use.pf.2Zoom`

This directory contains the code for the '2Zoom' application described in chapter 11.

Appendix B

This appendix describes a complete configuration file needed along with every applications designed in DLoVe's framework. The configuration file is called `config.txt` and contains ':' delimited fields. Comments can be specified in the configuration file. The '#' symbol indicates comment in the configuration file.

```
PORT NUMBER:      6326
Alarm Time:       1
Number of Alarms: 60

Drop Line:        3000
MODE:             MULTI_MESSAGING
GETE MODE:        NB_GETE
Workers Drop:     YES

Master Coordinator: u5.eecs.tufts.edu
Slave Coordinators: 1
Workers:          3

ID: u6.eecs.tufts.edu=1
ID: c15=2
ID: u8=3
ID: c13=4
```

PORT NUMBER This field indicated the port number the Master Coordinator binds and listens on for connections. All machines participating in a DLoVe distributed application need to know this port number.

Alarm Time This field is only used when a DLoVe program is compiled with the LVSTATS flag as described later. It indicates the frequency of alarms DLoVe will receive to dump internal data to standard output. When it is set to 2 for example, DLoVe will set the timer to receive an alarm every 2 seconds and will then print out various measurements described in chapter 13. If this variable is set to 0, then DLoVe will not receive any alarms at all even if it is compiled with the LVSTATS flag.

Number of Alarms In this field the programmer specifies the number of alarms DLoVe will receive before terminating. If it is set to 30, then after 30 alarms the application will terminate. This field is used together with conjunction with the Alarm Time field, which specifies the frequency of the alarms. This is used to make sure that all experiments execute the same amount time. If this field is set to 9999, then the application executes forever and it can receive alarms based upon the Alarm Time field.

Drop Line This field defines the 'drop' threshold variable. The number assigned to this variable determines how many pending

requests the Coordinator can tolerate before starting disregarding messages.

MODE

This field describes upon which protocol DLoVe will operate. Chapter 9 describes the 3 variations upon which DLoVe can operate. DLoVe can operate in single-messaging mode (SINGLE-MESSAGING) where each individual message such as SetE, GetE, Enable, and Disable are sent to the Workers from the Coordinator as single messages. Multi-messaging (MULTI-MESSAGING) allows DLoVe to send multiple messages to Workers. It is strongly recommended that this field is always set to MULTI-MESSAGING. SINGLE-MESSAGING has not been tested for a couple of years and it is not know if it still works. This field is used with conjunction with the next field described below.

GETE MODE

When DLoVe operates in multi-messaging, GetE requests can be defined to block or not to block for replies to come back from the Workers. When this field is set to NB_GETE, GetE requests are non-blocking, else they are, as described in chapter 9. It is strongly recommended that this field be always set to NB_GETE, because it is the optimum operation mode for DLoVe.

Workers Drop

If this field is set to YES, then the Workers also drop messages to implement the new modified algorithm described in chapter

13. It is strongly recommended that this field be always set to YES.

Master Coordinator This field must be set to the IP address or the DNS name where the Master Coordinator is executed.

Slave Coordinators This field indicates the number of Slave Coordinators participating in a multi-user Virtual Environment.

Workers This field indicates the number of Workers participating in a Virtual Environment.

ID This field may exist multiple times in the configuration file. It indicates the application layer ids of all Coordinators. For example, in a multi-user VR application with 3 Coordinators (one being the Master Coordinator) we could have:

```
ID: mondrian.eecs.tufts.edu=1
ID: monet=2
ID: 130.64.23.184=3
```

The IP address or the DNS name of the machines can be specified. Immediately after the IP address or the DNS name follows a '=' symbol and then the application layer id of that specific machine. DLoVe utilizes these application layer id number to associate devices with workstations, as described in chapters 8, 10, and 12.

Appendix C

The following code is a complete Virtual Reality program in DLoVe. A user uses an HMD to view the virtual world, and a Polhemus for a virtual hand with which he/she can grab and manipulate the 3D objects (program needs to be compiled with the source code defines HEAD and POLHEMUSCURSOR as shown in the code below. If this program is compiled with MOUSECURSOR and MOUSECOUPLER instead, the user can manipulate the camera and the virtual hand using the mouse.

```
// Choose (BOTH MOUSE...) OR POLHEMUSCURSOR
#define MOUSECURSOR 1
#define MOUSECOUPLER 1
// #define POLHEMUSCURSOR 1

// Choose this or not
// #define HEAD 1

#include "../pmiw.h"
#include "../opt/PfWindow.h"
#include "../basic/DeviceTimer.h"
#include "../eye/DeviceEye.h"

#include "../opt/HeadCoupler.h"
#include "../opt/MouseCoupler.h"
#include "../opt/PolhemusCursor.h"
#include "../opt/MouseCursor.h"
#include "../opt/Quitter.h"

#include <Performer/pf.h>
```

```

#include <Performer/pf/pfGroup.h>
#include <Performer/pf/pfScene.h>
#include <Performer/pf/pfGeode.h>
#include <Performer/pr/pfGeoSet.h>
#include <Performer/pf/pfChannel.h>
#include <Performer/pr/pfLight.h>
#include <Performer/pr/pfMaterial.h>
#include <Performer/pf/pfDCS.h>
#include <Performer/pfdu.h>
#include <Performer/pf/pfEarthSky.h>

#include "Grab4.h"

static PfWindow *pfwindow;
static DeviceTimer *deviceTimer;

pfLight *light;

CursorBase *cursor = NULL;
DeviceEye *deviceeye = NULL;
HeadCoupler *headcoupler = NULL;

static void makeGreen (pfGroup *parent) {
    const float ROOMSIZE = 1000000.f;    // Width and depth
    const float ROOMBOT = -2.0;

    pfGeoSet *gset = new pfGeoSet;
    gset->setPrimType(PFGS_QUADS);
    gset->setNumPrims(1);

    int i = 0;
    pfVec3 *scoords = (pfVec3*) new (4*sizeof(pfVec3)) pfMemory;
    scoords[i++].set (-ROOMSIZE/2., -ROOMSIZE, ROOMBOT);    // Front left
    scoords[i++].set (ROOMSIZE/2., -ROOMSIZE, ROOMBOT);    // Front right
    scoords[i++].set (ROOMSIZE/2., ROOMSIZE, ROOMBOT);    // back right
    scoords[i++].set (-ROOMSIZE/2., ROOMSIZE, ROOMBOT);    // back left

    gset->setAttr(PFGS_COORD3, PFGS_PER_VERTEX, scoords, NULL);

    pfVec3 *snorms = (pfVec3*) new (1*sizeof(pfVec3)) pfMemory;
    snorms[0].set( 0.0f, 0.0f, 1.0f);

    gset->setAttr(PFGS_NORMAL3, PFGS_PER_PRIM, snorms, NULL);

    pfVec4 *scolors = (pfVec4*) new (sizeof(pfVec4)) pfMemory;
    scolors[0].set (0.1, 0.7, 0.2, 1.);
    gset->setAttr(PFGS_COLOR4, PFGS_OVERALL, scolors, NULL);

    pfGeoState *gstate = new pfGeoState;
    gset->setGState (gstate);

    pfGeode *geode = new pfGeode;
    geode->addGSet (gset);
    parent->addChild (geode);
}

void InitializeHook() {

    pfGroup *root = new pfGroup;
    pfGroup *rootPickable = new pfGroup;
    root->addChild (rootPickable);

    #if POLHEMUSCURSOR || HEAD
        deviceeye = new DeviceEye ();
    #endif

    #ifndef HEAD
        headcoupler = new HeadCoupler (pfwindow);
    #endif

    #ifndef POLHEMUSCURSOR
        extern Variable<Pos6> *polhemus2;
    #endif
}

```

```

        cursor = new PolhemusCursor (root, rootPickable, polhemus2);
#endif

#ifdef MOUSECOUPLER
    (void) new MouseCoupler (pwindow);
#endif

#ifdef MOUSECURSOR
    cursor = new MouseCursor (root, rootPickable, pwindow);
#endif

//
// Make ground green (imagine you are in a park :)
//
makeGreen(root);

//
// Create 10 random objects that the user can grab and manipulate
//
const int NOBJS = 10;
Grab4 *g4[NOBJS];
int j;
for( j=0; j < NOBJS; j++) {
g4[j] = new Grab4(rootPickable, pfVec3 (RAND(100)-50 , RAND(50),
    RAND(10)), (int) RAND(4),
    pfVec4 (0.4+RAND(0.6), 0.4+RAND(0.6), 0.4+RAND(0.6), 1.0));
}

//
// Create the EarthSky (Performer fun)
//
pfEarthSky *esky = new pfEarthSky();
esky->setMode(PFES_BUFFER_CLEAR, PFES_SKY);
pwindow->GetChan()->setESky(esky);

pfScene *scene = new pfScene;
scene->addChild (root);
pwindow->GetChan()->setScene(scene);
pfNodePickSetup (scene);

//
// ESC will terminate application
//
(void) new Quitter (pfExit);
}

/*****
Window system initialization
*****/
static void InitializeWin () {
    pwindow = new PWindow(true, Link::GetHostId());
    deviceTimer = new DeviceTimer;

    pfEnable (PFEN_LIGHTING);

    light = new pfLight;
    light->setPos (0, -1., 1, 0.); // Tweak light position

    (new pfMaterial)->apply();
    pfLightModel *lmodel = new pfLightModel;
    lmodel->setAmbient(0.4, 0.4, 0.4); // Tweak light model (add more ambient)
    lmodel->apply();
}

/*****

```



```
*****/
main(int argc, char **argv) {
    Link::InitCommunication( (argv[1]) ? atoi(argv[1]) : 1);
    InitializeWin ();
    InitializeHook ();
    Link::InitSystem ();

    while (true) {
        Link::START();

        pfSync();
        if (headcoupler) headcoupler->UpdateManual();
        pfFrame();

        light->on();

        pfwindow->GetDeviceXWindow()->Read();
        if (deviceeye) deviceeye->Read ();
        deviceTimer->Read();

        pfwindow->GetDeviceXWindow()->Dispatch(&LinkStep::Step);

        if (deviceeye) deviceeye->Dispatch(&LinkStep::Step);
        deviceTimer->Dispatch();

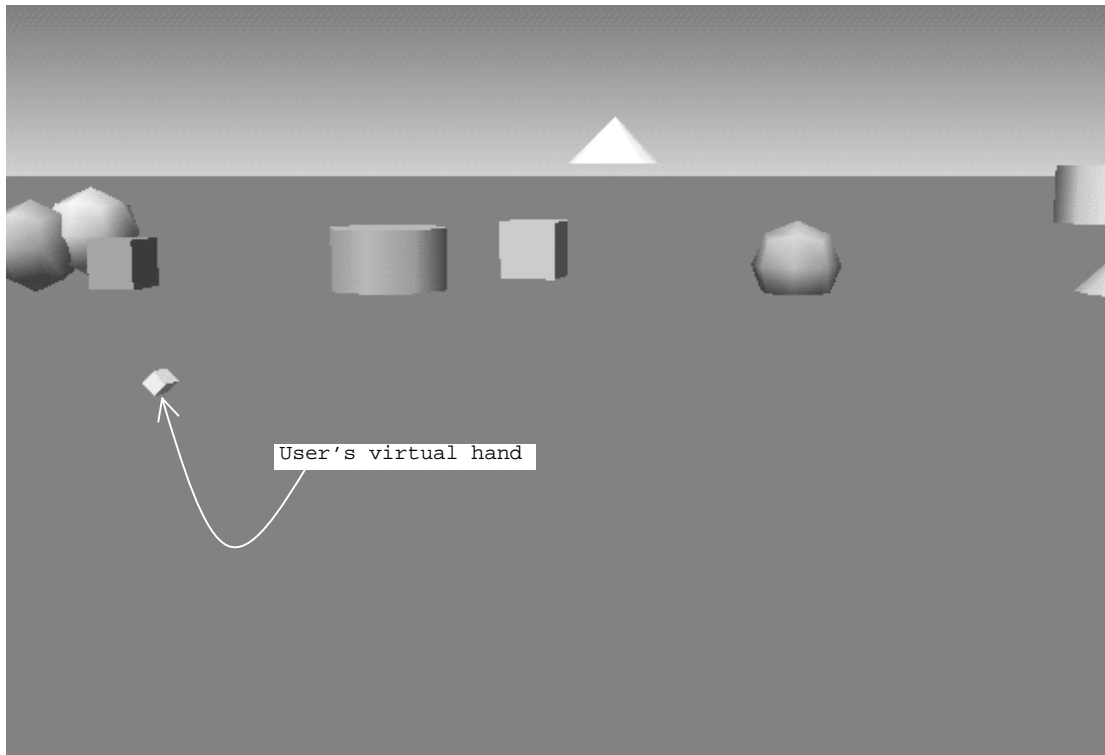
        cursor->DoHit();

        LinkStep::Step();
        IO::UpdateAll ();

        Link::FSTOP();
    }
}
```

When the same program is compiled first using the COORDINATOR and then the WORKER flag, two executables will be generated, one for the Coordinator and the other for the Worker(s), where the program can be now executed in a distributed environment.

The following picture is a snapshot of the program in execution. The various 3D objects are visible as well as the virtual hand of the user.



A simple VR program created using the DLoVe's paradigm

Appendix D

Installation instructions

DLoVe is located in the `implem` top directory. All sub-directories named `use.*` contain sample programs, and the other sub-directories contain the code for the UIMS, which one must compile together with his/her code.

All makefiles use the environment variable `HOSTTYPE`, which is set by the user's shell at login. If a system does not set it automatically, it has to be set by the user manually. I use "iris4d" when compiling on SGIs, and "sun4" when compiling on Suns.

Compatibility

The modules that do not use SGI Performer graphics compile using GNU g++ (ver. 2.7.2) on a Sun (Solaris ver. 5.5.1) or GNU g++ (ver. 2.7.2) or SGI CC (IRIX ver. 6.2)

on a Silicon Graphics (IRIX 6.2). The Performer code compiles only on SGI using the CC compiler, because the Performer libraries are generated with that compiler's name mangling conventions and cannot be linked under g++.

The basic UIMS only handles input, not output. It can read input from several sources, including an X window (a regular X window or an SGI mixed-mode Performer/GLX window), a Polhemus tracker, and an ISCAN eye tracker. One can add other devices by sub-classing DeviceBase with a new class that follows the same interface conventions, and then including calls to Read() and Dispatch() his device in the main loop.

Because we separate input event handling from graphics, the basic UIMS is compatible with any graphics system. We have used it with X and with SGI Performer. In addition, we provide optional classes and examples for use with X and Performer. If one is using one of those systems, these will be useful. If not, one must treat the UIMS as an input handling system, and provide his own connection to the graphics system.

Appendix E

Compile instructions

The makefiles in use.*/makefile perform all the necessary steps, but if one wants to modify it for his/her own applications or if he/she is not fluent in "makefiles", a description of the modules that need to be compiled is given below. The standard files one needs to compile to get the basic UIMS (these are defined as "OBJECTS" in the makefile) are shown below:

- basic/IO.cc
- basic/LinkBase.cc
- solve/Link.cc
- opt/Links.cc
- basic/VariableBase.cc
- solve/Variable.cc
- solve/Variables.cc
- basic/Condition.cc
- basic/EventHandler.cc
- basic/Pos.cc

- basic/utility.cc
- basic/DeviceXWindow.cc
- basic/DeviceTimer.cc

In addition, one would compile his/her main program (like use.x/example.cc) and his/her interaction objects (like use.x/XSlider.cc)

Building an X Windows application, one also wants to compile ("XOBJECTS" in the makefile):

- opt/XWindow.cc

If building a Performer application (that one can only do on an SGI), one generally wants to compile the following (which are "PFOBJECTS" in the makefile), in addition to the "OBJECTS" listed above:

- opt/PfVariables.cc
- opt/CursorBase.cc
- opt/MouseCursor.cc
- opt/PolhemusCursor.cc
- eye/DeviceEye.cc
- opt/HeadCoupler.cc
- opt/Quitter.cc
- opt/PfWindow.cc

Also, for Performer, the following libraries need to be included at the end of the CC command line (listed as "PFLIBS" in makefile):

-lpf -lpfutil -lmpc -limage -lGL -lXirisw -lfpe -lXmu -lX11 -lm -lmalloc -lC

If one writes state diagrams, he/she will need to run his/her .cc files through `state/{sun4,iris4d}/translate5` before compiling. `MouseCursor.cc` and `XSlider.cc` are examples of files that require this treatment (the makefile does this by creating `MouseCursor.cxx` first and then compiling the resulting .cxx files instead of the .cc)

Creating an application that will be executed in a distributed environment using Coordinator(s) and Worker(s), one must also compile the files in the `net` directory using the `COORDINATOR` and `WORKER` defines at the compiler's line. These files are listed in all makefiles as `DLOVE_OBJECTS`. These files are listed below:

- `net/BasicComm.cc`
- `net/Stats.cc`
- `net/Config.cc`
- `net/Top.cc`
- `net/Connect.cc`
- `net/Coordinator.cc`

The `Stats.cc` file contains modules responsible for printing out raw data when measuring performance. The files `BasicComm.cc`, and `Top.cc` are the super-classes of the Coordinator and the Worker. The `Coordinator.cc` contains all the code for making the Coordinator accept and connections from Workers, and when multiple users are involved, this module also directs Workers to connect to the other Coordinators (Slave Coordinators). The `Config.cc` is a module used for reading and parsing the configuration file.

Compiler flags

Compiling on Suns using the Gnu g++ compiler the following flags must be at the g++ command line:

```
-I/usr/openwin/include -fno-implicit-templates
```

Compiling on SGIs using the CC compiler the following flag must be in the CC command line:

```
-no_auto_include
```

The makefiles look for the HOSTTYPE environment variable to find the executable for translate5 and to include a host-specific makefile such as basic/makefile.\$HOSTTYPE. If normal login does not set this environment variable, one must set it manually. We use "iris4d" when compiling on SGIs, and "sun4" when compiling on Suns. The makefile for sun4 also looks for the environment variable OPENWINHOME and one can change or delete this to suit his/her compilation.

Compilation for distributed environments

All DLoVe's source files need to be recompiled using the following two flags in order to for the user to obtain two executables, one for the Coordinator(s) and one for the Worker(s).

- COORDINATOR
- WORKER

Other flags with which DLoVe can be compiled are:

- LVSTATS
- INFO
- MULTIUSER

The LVSTATS flag is used for DLoVe to generate raw data that can be analyzed to measure its performance. Programs that can be used to measure its performance are supplied in the `scripts` directory. The INFO flag is used for debugging purposes. DLoVe prints out the data structure of all Links and Variables, types of messages exchanged between the Coordinator(s) and the Worker(s), and the output of the partition algorithm. The MULTIUSER flag is used when compiling multi-user programs. When using this flag, either the COORDINATOR or the WORKER flag must also be specified on the compiler's command line.

Bibliography

[**Adam 93**] John A. Adam. "Virtual Reality is for Real", IEEE Spectrum Oct. 1993

[**Akl 97**] Akl, Selim G. "Parallel Computation Models and Methods". New Jersey: Prentice Hall PTR, 1997.

[**Alias**] Alias Research Inc., "ALIAS User's Guide" 110 Richmond Street East, Toronto Ontario, Canada M5C 1P1

[**Amdahl 67**] Amdahl, G.M. Validity of the single-processor approach to achieving large scale computing capabilities. In AFIPS Conference Proceedings vol. 30 (Atlantic City, N.J., Apr. 18-20). AFIPS Press, Reston, Va., 1967, pp. 483-485.

[**Anderson 95**] David B. Anderson, John W. Barrus, John H. Howard, Charles Rich, Chia Shen, Richard C. Waters. "Building Multi-User Interactive Multimedia Environments at MERL", IEEE MultiMedia, 2(4):77-82, Winter 1995.

[**Aukstakalnis 92**] Steve Aukstakalnis, David Blatner, "Silicon Mirage: the art and science of virtual reality", Peachpit Press, 1992

[**Bach 86**] Bach, Maurice J. "The Design of the UNIX Operating System". New Jersey: Prentice Hall PTR, 1986.

[**Barbosa 96**] Barbosa, Valmir C. "An Introduction to Distributed Algorithms". Massachusetts Institute of Technology, 1996.

[**Barfield 95**] Barfield, Woodrow and Claudia Hendrix. "Factors Affecting Presence and Performance in Virtual Environments." In Interactive Technology and the New Paradigm for Healthcare. Washington, DC: IOS Press. 1995 p21-28

[**Berson 96**] Berson, Alex. "Client/Server Architecture". Second Edition. McGraw-Hill Companies, 1996.

[**Bharat 94**] Krishna Bharat, and Marc H. Brown., "Building Distributed Multi-User Applications by Direct Manipulation", UIST '94, November 2-4 1994 pages 71-81

[**Bharat 97**] Kurani, Bharat. "Applied UNIX Programming vol 2". New Jersey: Prentice Hall PTR, 1997.

[**Bier 86**] Bier, E., and M. Stone, "Snap-Dragging," SIGGRAPH 86, 233-240

[**Borning 81**] Borning A. 1981. The programming language Aspects of ThingLab; a constraint-oriented simulation laboratory. ACM Trans. Programming. Lang. Syst. 3,4 (Oct) p353-387

[Borning 86] Alan Borning and Robert Duisberg, Constraint-Based Tools for Building User Interfaces. ACM Transactions on Graphics, Vol. 5, No. 4, October 1986, pages 345-374

[Bowman 95] Bowman, Duane K. "International Survey: Virtual-Environment Research". IEEE Computer, (June) 1995, p56-65

[Bran 94] Selic, Bran and Gullekson, Garth and Ward, Paul T. "Real Time Object Oriented Modeling". John Wiley & Sons, Inc., 1994.

[Bräunl 89] T. Bräunl, "Structured SIMD Programming in Parallaxis, Structured Programming", vol. 10, no. 3, July 1989, pp. 121-132 (12)

[Bräunl 91] T. Bräunl, "Designing Massively Parallel Algorithms with Parallaxis", Proceedings of the 15th Annual International Computer Software & Applications Conference, compsac91, Sep. 1991, pp. 612-617 (6)

[Burdea 94] Grigore Burdea, and Philippe Coiffet, "Virtual Reality Technology", Wiley-Interscience Publication 1994

[Buyya 99a] Rajkuma Buyya, "High Performance Cluster Computing" Vol. 1 (Architectures and Systems), Prentice Hall PTR, New Jersey 1999

[Buyya 99b] Rajkuma Buyya, "High Performance Cluster Computing" Vol. 2 (Programming and Applications), Prentice Hall PTR, New Jersey 1999

[**Carlsson 93**] Carlsson, Christer, and Olaf Hafsand, "DIVE: A Multi-User Virtual Reality System". Proceedings of the IEEE Virtual Reality Annual International Symposium, Sep. 18-22, 1993, p394-401

[**Carlsson 93**] Carlsson, C., and O. Hagsand. "DIVE - A platform for multi-user virtual environments". Computers and Graphics 17(6): 663-669, 1993

[**Casner 94**] Casner Steve, Henning Schulzrinne, and David M. Kristol "Frequently Asked Questions (FAQ) on the Multicast Backbone (MBONE)", 1994, <ftp://venera.isi.edu/mbone/faq.txt>

[**Clark 90**] Clark D. D. and D. L. Tennenhouse , "Architectural considerations for a new generation protocols", SIGCOMM 1990, Sep. 1990, Computer Communication Review, 20(4), 200-208

[**Cormen 90**] Cormen, Thomas H. and Leiserson, Charles E. and Rivest, Ronald L. "Introduction to Algorithms". Massachusetts Instritution of Technology, 1990.

[**Coulouris 95**] Coulouris, George and Dollimore, Jean and Kindberg, Tim. "Distributed Systems Concepts and Desing second edition". Addition-Wesley Publishers Ltd., 1995.

[**Crowcroft 95**] Jon Crowcroft, "Open Distributed Systems", Artech House, Inc., MA 1995.

[Culler 99] Culler, David E. and Singh, Jaswinder Pal. "Parallel Computer Architecture A Hardware Software Approach". Morgan Kaufmann Publishers, Inc., 1999.

[Curry 96] Curry, David A. "UNIX Systems Programming for SVR4". O'Reilly & Associates, Inc., 1996.

[Deering 92] Michael Deering, "High Resolution Virtual Reality" Computer Graphics, 26,2. July 1992.

[DIVE] DIVE Web site: <http://www.sics.se/dce/dive/dive.html>

[Dowd 98] Kevin Dowd & Charles R. Severance, "High Performance Computing" (Second Edition) RISC Architectures, Optimization & Benchmarks. O'Reilly & Associates, Inc, July 1998

[Doyle 95] Doyle, Werner K. Interactive "Image-Directed Epilepsy Surgery: Rudimentary Virtual Reality in Neurosurgery". In Interactive Technology and the New Paradigm for Healthcare. Washington, DC: IOS Press, 1995 p. 91-100

[Elliott 94] C. Elliott, G. Schechter, R. Yeung and S. Abi-Ezzi. "TBAG: A High Level Framework for Interactive, Animated 3D Graphics Applications", In Proc. ACM SIGGRAPH '94, pages 421-434, August, 1994.

[Flynn 66] M. J. Flynn, "Very high speed computing systems", Proc. IEEE 54 pp 1901-9 (1966).

[Foley 87] J.D. Foley, "Interfaces for Advanced Computing", Scientific American, v257, n4 p127-135, October 1987.

[Fountains 94] T. J. Fountains, "Parallel Computing principles and practice", Press Syndicate of the University of Cambridge 1994.

[Freeman-Benson 90] Bjorn N. Freeman-Benson, John Maloney, and Alan Borning. An Incremental Constraint Solver. Communications of the ACM, Vol. 33 Number 1, Jan. 1990 p54-63.

[GL 91] Silicon Graphics Inc. Graphics Library Programming Guide, 1991.

[Gleicher 93] Michael Gleicher. "A Graphical Toolkit Based on Differential Constraints". UIST '93 November 1993, pages 109-120.

[Gobbetti 93] Enrico Gobbetti and Jean-Francis Balaguer. "VB2: An Architecture for Interaction in Synthetic Worlds". In Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology, pages 167-178, Atlanta, Georgia, November 1993.

[Gray 97] Gray, John Shapley. "Interprocess Communications in UNIX The Nooks and Crannies". New Jersey: Prentice Hall PTR, 1997.

[Green 91] M. Green, and R.J.K. Jacob, "Software Architectures and Metaphors for Non-WIMP User Interfaces", Computer Graphics, v25, n3 p229-235, July 1991.

[Greenhalgh 95a] Greenhalgh, C., and Benford, S., “MASSIVE: a Distributed Virtual Reality System Incorporating Spatial Trading,” in Proc. IEEE 15th International Conference on Distributed Computing Systems (DCS’95), Vancouver, Canada, May 30 – June 2, 1995, IEEE Computer Society.

[Greenhalgh 95b] Greenhalgh, C., and Benford, S., “MASSIVE: A Virtual Reality System for Tele-conferencing”, ACM Transactions on Computer Human Interfaces (TOCHI), 2 (3), pp. 239-261, ISSN 1073-0516, ACM Press, Sep. 1995.

[Gross 98] Gross, Thomas and O’Hallaron, David R. “iWarp Anatomy of a Parallel Computing System”. Massachusetts Institute of Technology, 1998.

[Gupta 93] A. Gupta, A. Grama, and V. Kumar. “Isoefficiency: Measuring the Scalability of Parallel Algorithms and Architectures”. IEEE Parallel and Distributed Technology, p12-20, August 1993.

[Gustafson 88a] John L. Gustafson, Gary R. Montry, and Robert E. Benner. Development of Parallel Methods for 1024-Processor Hypercube. SIAM Journal on Scientific and Statistical Computing, 9(4), July 1988.

[Gustafson 88b] J. L. Gustafson, “Reevaluating Amdahl’s Law” chapter book, Supercomputers and Artificial Intelligence, Edited by Kai Hwang, 1988.

[Gustafson 90] J. L. Gustafson, “Fixed Time, Tiered Memory, and Superlinear Speedup”, Proceedings of the Fifth Distributed Memory Computing Conference (DMCC5), October 1990.

[Hagsand 96] Hagsand, O. "Interactive multiuser VEs in the DIVE system". IEEE Multimedia 3(1): 30-39, 1996.

[Halabi 97] Bassam Halabi, "Internet Routing Architectures, The definite resource for internetworking design alternatives", New Riders Publishing 1997.

[Halliday 94] Sean Halliday, and Mark Green. "A Geometric Modeling and Animation System for Virtual Reality". Virtual Reality Software & Technology, Proceedings of the VRST '94 Conference, 23-26 August 1994, Singapore pages 71 - 84. Published by World Scientific Publishing Co. Pte. Ltd.

[Hesham 94] Hesham El-Rewini, Theodore G. Lewis, and Hesham H. Ali. "Task Scheduling in Parallel and Distributed Systems". PTR Prentice Hall, Prentice-Hall, Inc., Englewood Cliffs, New Jersey 1994.

[Hill 92] Raph D. Hill, "The Abstraction-Link-View paradigm: Using constraints to connect user interfaces to applications". CHI '92, May 3-7, 1992 pages 335-342.

[Hill 93] Hill, R. D. 1993. "The Rendezvous constraint maintenance system". In ACM SIGGRAPH Symposium on User Interface Software and Technology, Proceedings UIST'93. ACM, New York, p225-234.

[Hill 94] Ralph D. Hill, Tom Brinck, Steven L. Rohall, John F. Patterson, and Wayne Wilner, "The Rendezvous Architecture and Language for Constructing Multiuser Applications". Transactions on Computer-Human Interaction v1,n2 Jun 1994 p81-125.

[Hodges 95] Hodges, Larry F., Rob Kooper, Thomas C. Meyer, Barbara O. Rothbaum, Dan Opdyke, Johannes J. de Graaff, James S. Willford, and Max M. North. "Virtual Environments for Treating the Fear of Heights." *IEEE Computer* 28,7 (July) 1995 p27-34.

[Holbrook 95] Hugh W. Holbrook, Sandeep K. Singhal, and David R. Cheriton, "Log-Based Receiver-Reliable Multicast for Distributed Interactive Simulation", In Proceedings of ACM SIGCOMM Aug. 1995 p328-341. Published as *Computer Communications Review*, Vol 25, No. 4 October 1995.

[Hoover 87] Hoover R. "Incremental graph evaluation". Ph.D. thesis, Dept. Of Computer Science, Cornell U., Ithaca, NY 1987.

[Horn 92a] Bruce Horn. "Constrained patterns as a basis for object-oriented constraint programming". In Proceedings of the 1992 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications, pages 218-233, Vancouver, British Columbia, October 1992.

[Horn 92b] Bruce Horn. "Properties of user interface systems and the Siri programming language". In Brad Myers, editor, *Languages for Developing User Interfaces*, pages 211-236. Jones and Bartlett, Boston, 1992.

[Hudson 91] Hudson, A., "Incremental Attribute Evaluation: A Flexible Algorithm for Lazy Update", *ACM Transactions on Programming Languages and Systems*, v13, n3, July 1991, pp. 315-341.

[Hwang 98] Hwang, Kai and Xu, Zhiwei. "Scalable Parallel Computing". McGraw-Hill, 1998.

[Jamieson 87] Jamieson, Leah H. and Gannon, Dennis B. and Douglass, Robert J. "The Characteristics of Parallel Algorithms". Massachusetts Institute of Technology, 1987.

[Jeffrey 96] Jeffrey J. P. Tsai, Yaodong Bi, Steve J. H. Yang, and Ross A. W. Smith, "Distributed Real-Time Systems", John Wiley & Sons, Inc. 1996.

[Johnston 92] R.S. Johnston, "The SimNet Visual System," Proc. 9th Interservice/Industry Training Equipment Conf., Defense Technical Information Center, Washington. DC. Nov. 1992.

[Kelsick 98] Kelsick, Jason and Vance, Judy M., "The VR Factory: Discrete Event Simulation Implemented in Virtual Environment", ASME Design for Manufacturing Conference Proceedings, Atlanta, GA, Sep 1998.

[Kemelmakher 98] M. Kemelmakher and O. Kremien. "Scalable and Adaptive Resource Sharing in PVM. Recent Advances in Parallel Virtual Machine and Message Passing Interface". LNCS, Vol 1479, p196-205, Springer-Verlag 1998.

[Kenneth 97] Berman, Kenneth A. and Paul, Jerome L. "Fundamentals of Sequential and Parallel Algorithms". PWS Publishing Company, 1997.

[Kernigham 84] Kernigham, Brian W. and Pike, Rob. "The UNIX Programming Environment". New Jersey: Bell Telephone Laboratories, 1984.

[Krueger 94] Wolfgang Krueger and Bernd Froehlich, "The Responsive Workbench", IEEE Computer Graphics and Applications, May 1994.

[Lakshmivarahan 90] Lakshmivarahan, S. and Dhall, Sudarshan K. "Analysis and Design of Parallel Algorithms. Arithmetic and Matrix Problems". McGraw-Hill, 1990.

[Langreth 95] Langreth, Robert. "Virtual Reality: Breakthrough Hand Controller" Popular Science 246, January 1995 p 45.

[Larson 92] James A. Larson, "Interactive Software: Tools for Building Interactive User Interfaces". Yourdon Press, by Prentice-Hall 1992, Englewood Cliffs, New Jersey 07632.

[Lewis 98] Chris Lewis, "Cisco TCP/IP Routing Professional Reference (second edition)", McGraw-Hill Companies, Inc. 1998.

[Lopez 94a] Gus Lopez, Bjorn Freeman-Benson, and Alan Borning, "Kaleidoscope: A Constraint Imperative Programming Language", In Constraint Programming, B. Mayoh, E. Tougu, J. Penjam (Eds.), NATO Advance Science Institute Series, Series F: Computer and System Sciences, Vol. 131, Springer-Verlag, 1994, pages 313-329.

[Lopez 94b] Gus Lopez, Bjorn Freeman-Benson, and Alan Borning, "Implementing Constraint Imperative Programming Languages: the Kaleidoscope'93 Virtual Machine", System, Languages, and Applications, Portland, Oregon, October 1994, pages 259-271.

[Lynch 96] Lynch, Nancy A. "Distributed Algorithms". Morgan Kaufmann Publishers, Inc., 1996.

[Macedonia 95a] Macedonia, Michael, R. Michael J. Zyda, David R. Pratt, and Paul T. Barham, "Exploiting Reality with Multicast Groups: A Network Architecture for Large Scale Virtual Environments", Proceedings of IEEE Virtual Reality Annual International Symposium, 2-10, 1995.

[Macedonia 95b] Macedonia, Michael R., Zyda, Michael J., Pratt, David R., Brutzman, Donald P. and Barham, Paul T., "Exploiting Reality with Multicast Groups", IEEE Computer Graphics & Applications Sep. 1995 p38-45.

[Macedonia 95c] Macedonia, Michael R., Brutzman, Donald P., Zyda, Michael J., Pratt, David R., Barham, Paul T., Falby, John and Locke, John, "NPSNET: A Multi-Player 3D Virtual Environment Over the Internet" in the Proceedings of the 1995 Symposium on Interactive 3D Graphics, 9-12 April 1995, Monterey, CA.

[Macedonia 97] Macedonia, Michael and Zyda, Michael, "A Taxonomy for Networked Virtual Environments", IEEE Multimedia, v4, n1, January-March 1997, p48-56.

[MacIntyre 98] Brair MacIntyre and Steven Feiner. "A Distributed 3D Graphics Library". Proc. ACM SIGGRAPH '98 Conference, Addison-Wesley/ACM Press, 1998.

[MPI 94] MPI Message Passing Interface Forum. "MPI: A Message-Passing Interface Standard". International Journal of Supercomputer Applications and High Performance Computing, 8(3/4), 1994.

[**Mullender 93**] Mullender, Sape. "Distributed Systems second edition". ACM Press Frontier Series, 1993.

[**Myers 90a**] B. A. Myers, D. A. Giuse, R. B. Dannenberg, B. V. Zanden, D. S. Kosbie, E. Pervin, A. Mickish, and P. Marchal. "Garnet: Comprehensive Support for Graphical, Highly Interactive User Interfaces.", IEEE Computer Vol. 23, No11, November 1990, p 71-85.

[**Myers 90b**] B.A. Myers et al., "The Garnet Toolkit Reference Manuals: Support for highly Interactive, Graphical User Interfaces in Lisp." Tech. Report CMU-CS-90-117, Carnegie Mellon University, Computer Science Department, Mar. 1990.

[**Myers 92a**] Myers, B. A., and Zanden, B. Vander 1992. "An environment for rapid creation of interactive design tools". Vis. Comput. Int. Comput. Graph. 8, 3, 94-116.

[**Myers 92b**] Myers, B. A., Giuse, D. A., and Vander Zanden, B. 1992. "Declarative programming in a prototype-instance system: Object-oriented programming without writing methods". Sigplan Not. 27, 10 (Oct.), pages 184-200.

[**Pacheco 97a**] Peter S. Pacheco, Parallel "Programming with MPI", Morgan Kaufman Publishers, Inc. 1997.

[**Pacheco 97b**] Pacheco, Peter S. "Parallel Programming with MPI". Morgan Kaufmann Publishers, Inc., 1997.

[Pankaj 94] Jalote, Pankaj. "Fault Tolerance in Distributed Systems". PTR Prentice Hall, 1994.

[PARADISE] PARADISE Project Web site: <http://www.dsg.stanford.edu/paradice.html>

[Pate 96] Pate, Steve D. "UNIX Internals. A Practice Approach". Addison Wesley Longman, 1996.

[PHIGS 88] PHIGS+ Committe. Phigs+ functional description, revision 3.0. Computer Graphics.22(3):125-125, 1988.

[Pimentel 94] Ken Pimentel, and Brian Blau, "Teaching Your System To Share", IEEE Computer Graphics & Applications, Jan 1994, pages 60-65.

[Prat 95] Pratt, David R., Michael Zyda, and Kristen Kelleher. "Virtual Reality: In the Mind of the Beholder." IEEE Computer 28,7 (July) 1995 p17-19.

[Rago 93] Rago, Stephen A. "UNIX System V Network Programming". Addison - Wesley Publishing Company, Inc., 1993.

[Reason 78] Reason JT, "Motion sickness adaptation: a neural mismatch model". Journal of the Royal Society of Medicine, 71, p819-829, 1978.

[Rewini 98] Hesham El-Rewini & Ted G. Lewis, "Distributed and Parallel Computing". Mannining Publications Co. 1998.

[Robbins 96] Robbins, Kay A. and Robbins, Steven. "Practical UNIX Programming. A Guide to Concurrency, Communication, and Multithreading". New Jersey: Prentice Hall PTR, 1996.

[Rochkind 95] Rochkind, Marc J. "Advanced UNIX Programming". New Jersey: Prentice Hall PTR, 1995.

[Rohlf 94] J. Rohlf and J. Helman, IRIS Performer: A High Performance Multiprocessing Toolkit for Real-Time (3D) Graphics, In Proc. ACM SIGGRAPH '94, pages 381-394, 1994.

[Sannella 92] Sannella, M. And Borning, A. 1992. "Multi-Garnet: Integrating multi-way constraints with Garnet". Tech Rep. 92-70-01, Dept. Of Computer Science and Engineering, Univ. Of Washington, Seattle, Wash.

[Sannella 94] Michael Sannella. "SkyBlue: A Multi-Way Local Propagation Constraint Solver for User Interface Construction". UIST '94 November 2-4, 1994, pages 137-146.

[Satava 93] Satava, Richard M. "Virtual Reality Surgical Simulator: The First Steps" Surgical Endoscopy 7, 1993 p203-05 and in VR93: Proceedings of the Third Annual Conference on Virtual Reality, London, April 1993. London: Meckler Ltd. pp103-05.

[Schroder 94] Schroder-Preikschat, Wolfgang. "The Logical Design of Parallel Operating Systems". New Jersey: Prentice Hall PTR, 1994.

[Shaw 93] Shaw, Chris, and Mark Green, "The MR Toolkit Peers Package and Experiment". Proceedings of the IEEE Virtual Reality Annual International Symposium, 1993, p463-469.

[Shneiderman 92] B. Shneiderman, "Designing the User Interface: Strategies for Effective Human-Computer Interaction", Second Edition, Addison-Wesley, Reading, MA, 1992.

[Shoch 82] Shoch, J. F. and J. A. Hupp, "The Worm programs - early experience with a distributed computation", Communications of the ACM 25(3) 1982.

[Singhal 94] Sandeep K. Singhal, and David R. Cheriton, "Using a Position History-Based Protocol for Distributed Object", Technical Report STAN-CS-TR-94-1505, Dep. Of Computer Science, Stanford University, Feb. 1994.

[Singhal 95] Sandeep K. Singhal, and David R. Cheriton, "Exploiting position history for efficient remote rendering in networked virtual reality", Presence: Teleoperators and Virtual Environments, 4(2) p169-193, Spring 1995.

[Singhal 96] Singhal, S. K., and D. R. Cheriton. "Using projection aggregations to support scalability in distributed simulation". In Proceedings of the 16th International Conference on Distributed Systems (ICDCS), 196-206. IEEE Computer Society, May 1996.

[Singhal 99] Sandeep Singhal and Michael Zyda, "Networked Virtual Environments" Design and Implementation, Addison-Wesley, ACM Press, New York NY 1999.

[Slattery 99] Terry Slattery, Bill Burton, “Advanced IP Routing in Cisco Networks”, McGraw-Hill Companies, Inc. 1999.

[Sowizral 98] H. Sowizral, K. Rushforth, and M. Deering. The Java 3D API Specification, Addison-Wesley, Reading, MA, 1998.

[Spurgeon 96] Spurgeon, Charles. “Ethernet Configuration Guidelines: A Quick Reference Guide to the Official Ethernet (IEEE 802.3) Configuration Rules”. Includes 100Base-T Fast Ethernet. Charles Spurgeon, 1996.

[Stephen 94] Ellis, Stephen R. “What Are Virtual Environments?” IEEE Computer Graphics and Application v14 no1 January 1994.

[Stevens 90] Stevens, Richard W. “UNIX Network Programming”. New Jersey: PTR Prentice Hall, 1990.

[Stevens 92] Stevens, Richard W. “Advanced Programming in the UNIX Environment”. Addison Wesley Publishing Company, 1992.

[Stevens 98] Stevens, Richard W. “UNIX Network Programming”. Networking APIs: Sockets and XTI. Vol 1 second edition. New Jersey: Prentice Hall PTR, 1998.

[Strauss 92a] Paul S. Strauss and Rikk Carey. An object-oriented 3d graphics toolkit. Computer Graphics, 26(2):341-349, July 1992. Proceedings SIGGRAPH '92.

[Strauss 92b] P.S. Strauss and R. Carey, An Object-Oriented 3D Graphics Toolkit, In Computer Graphics (Proc. ACM SIGGRAPH '92), pages 341-349 Aug, 1992.

[**Sturman 92**] D.J. Sturman, "Whole-Hand Input", doctoral dissertation, Media Lab, Massachusetts Institute of Technology, Cambridge, MA., Feb. 1992.

[**Sunderam 90**] V. S. Sunderam, PVM: A Framework for Parallel Distributed Computing, *Concurrency: Practice and Experience*, 2, 4, pp 315--339, December, 1990. Also at <http://www.netlib.org/ncwn/pvmsystem.ps>

[**Sutherland 63**] Ivan Sutherland. "Sketchpad: A Man Machine Graphical Communication System". Ph.D thesis, Massachusetts Institute of Technology, January 1963.

[**Szekely 88**] Pedro A. Szekely and Brad A. Myers. A user interface toolkit based on graphical objects and constraints. In *OOP-SLA '88 Proceedings*, pages 36-45, September 1989. Also as *Sigplan Not.* 23, 11 (Nov) 1988 p36-45.

[**Tagg 97**] Roger Tagg and Chris Freyberg, "Designing Distributed and Cooperative Information Systems", International Thomson Computer Press, MA 1997.

[**Tanenbaum 92**] Tanenbaum, Andrew S. "Modern Operating Systems". New Jersey: Prentice Hall PTR, 1992.

[**Tanenbaum 95**] Tanenbaum, Andrew S. "Distributed Operating Systems". New Jersey: Prentice Hall PTR, 1995.

[**Tanenbaum 96**] Andrew S. Tanenbaum, "Computer Networks" (third edition), Prentice Hall PTR, New Jersey 1996.

[**Terrence 97**] Chan, Terrence. "UNIX System Programming Using C++". New Jersey: Prentice Hall PTR, 1997.

[**Utpal 88**] Banerjee, Utpal. "Dependence Analysis for Supercomputing". Kluwer Academic Publishers, 1988.

[**Vahalia 96**] Vahalia, Uresh. "UNIX Internals The New Frontiers". New Jersey: Prentice Hall PTR, 1996.

[**Vince 95**] John Vince, "Virtual Reality Systems", Addison-Wesley Publishing Company 1995.

[**Waters 96**] Richard C. Waters, David B. Anderson, John W. Barrus, David C. Brogan, Michael A. Casey, Stephan G. McKeown, Tohei Nitta, Ilene B. Sterns, William S. Yerazunis, "Diamond Park and Spline: A Social Virtual Reality System with 3D Animation", Spoken Interactions, and Runtime Modifiability. Presence: Teleoperators and Virtual Environments, Nov 1996.

[**Wilson 95**] Wilson, Gregory V. "Practical Parallel Programming". Massachusetts Institute of Technology, 1995.

[**Wilson 96**] Wilson, Gregory V. and Lu, Paul. "Parallel Programming Using C++", Massachusetts Institution of Technology, 1996.

[Zanden 94] Brad Vander Zanden, Brad A. Myers, Dario A. Guise, Pedro Szekely., "Integrating Pointer Variables into One-Way Constraint Models". ACM Transactions on Computer-Human Interaction, v1, n2 June 1994 p161-213.

[Zimmerman 87] T. G. Zimmerman et al., "A Hand Gesture Interface Device." Proc. Human Factors in Computing Systems and Graphics Interface, ACM Press, New York, April 1987, pp. 189-192.

[Zyda 93] Zyda, Michael J., Pratt, David R., John S. Falby, Chuck Lombardo, Kelleher, Kristen M. "The Software Required for the Computer Generation of Virtual Environments". In Presence. 2,2 (Spring 1993). p130-140.