

Concur: An Investigation of Lightweight Migration in Support of Centralized Synchronous Distributed Collaboration

John Edward Menges

A dissertation submitted to the faculty of the University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science.

Chapel Hill
2009

Approved by:

Kevin Jeffay

F. Donelson Smith

John B. Smith

David Stotts

Stephen F. Weiss

© 2009
John Edward Menges
ALL RIGHTS RESERVED

Abstract

John Edward Menges: Concur: An Investigation of Lightweight Migration in Support of Centralized Synchronous Distributed Collaboration.

(Under the direction of Kevin Jeffay)

Synchronous distributed collaborative systems support simultaneous observation of and interaction with shared objects by multiple, dispersed participants. Centralized architectures for such systems support simple user mental models and are comparatively easy to implement, but they suffer from high latency. Replicated architectures improve latency at the expense of more complex user mental models and implementations. Hybrid and dynamic architectures apply centralized and replicated sub-architectures in an attempt to get the best of both worlds, but in reality they further complicate implementations and user mental models.

Concur is an architecture I developed to investigate lightweight migration as an alternative means to attain the best characteristics of centralized and replicated architectures. Previous dynamic architectures improved latency using the migration of heavyweight processes or (object-oriented) objects, which is costly in terms of migration time and runtime requirements. In Concur I have instead organized collaborative applications and supporting infrastructures around migrating *entities* which are not required to have full process or object semantics. These entities are classified by properties affecting migration, such as their size and their use of external references. In this way I achieved both the simpler user mental models and implementations of centralized systems and the superior latency characteristics of replicated systems. Concur accomplishes this through the fast migration of lightweight entities in a *multi-centered* centralized system, where a multi-centered system is defined as having single physical center and multiple, independently-migrating and entity-specific logical centers.

This dissertation also identifies other significant advantages of the Concur architecture. Sub-object, easily-migratable entity classes minimize runtime requirements, facilitating widespread entity distribution. This in turn helps us to achieve the critical mass required for the success of any communication technology. The speed of lightweight entity migration also enables migrations to be triggered based on telegraphed user intentions (user actions that hint at imminent succeeding actions). I have demonstrated that telegraphed intentions are more accurate predictors of future interactions than the recent interaction histories considered in previous systems. Migrating entities based on these telegraphed intentions increases the probability that an entity will be located near a user when he begins to manipulate it.

To Nancy

Acknowledgments

I would like to thank my advisor, Kevin Jeffay, for his patience, his encouragement to pursue these ideas, his confidence in my work, for continually asking me to do a little more or a little better, and for his new perspectives that often seemed wrong to me at first but hardly ever were.

To my other committee members, who knew what I ought to know and made sure I learned it, and who always picked up on the important parts I'd missed, I'm appreciative. They are Don Smith, John Smith, David Stotts, and Steve Weiss.

My fellow "gradual student" Dr. Rajeev Pandey was enormously helpful as he prodded me to finish and kept reminding me that I could do it. Jay Aikat cheerfully provided lots of valuable assistance with the experimental laboratory.

I am thankful for my late father's confidence in me, and his willing support and constant encouragement. My mother died before I began graduate school, but she gave of herself to me for many years in ways that will be with me forever.

My children, Nathaniel, Lewis, Amelia, Michael, and Brian, have been precious diversions for me. I am glad that I was able to finish school before you did!

To my wife Nancy, who patiently and cheerfully endured too many years of my schooling and was always more confident in me than I was in myself, I am especially grateful. I love you!

Finally, to God, who made such an interesting world of things and ideas to explore, thank you! I am in awe of you!

Table of Contents

List of Tables	xii
List of Figures	xiii
List of Abbreviations	xxi
List of Symbols	xxii
1 Introduction, Definitions, and Overview	1
1.1 Collaboration and Virtual Things	1
1.2 Virtual Representations of Real Things	3
1.3 Synchronous Distributed Collaboration	8
1.4 Separation of Models and Views	10
1.5 The Model-View-Controller Paradigm and the Observer Design Pattern	13
1.6 Centralized and Replicated Architectures	20
1.7 Hybrid Architectures and Dynamic Reconfiguration	30
1.8 Problem Statement and Thesis	30
1.9 Contributions of this Work	35
1.10 Evaluation Summary	37
1.10.1 Experiment Setup	38
1.10.2 Notes on Experimental Result Plots in this Dissertation	40
1.10.3 Experimental Results	42

1.11	Dissertation Outline	53
2	Related Work	60
2.1	Introduction	60
2.2	Example Centralized Synchronous Distributed Collaborative Systems	61
2.2.1	XTV and Chung's Logging Infrastructure	61
2.2.2	Rendezvous	65
2.2.3	Weasel and Clock	67
2.3	Issues with the Above Systems	72
2.3.1	Functionality	73
2.3.2	Implementation	76
2.3.3	Performance	80
2.4	Contributions of this Work	81
2.5	Analysis Framework for Centralized Synchronous Distributed Collaborative Systems	83
2.5.1	Models	84
2.5.2	Protocol Manipulators	87
2.5.3	View Computation Engines	88
2.5.4	Local View State Repositories	90
2.5.5	View Specifications	91
2.5.6	View Realizations	92
2.5.7	Controllers	92
2.5.8	Analysis Summary	93
2.6	Other Related Work	95
2.6.1	Perspective-Like Constructs and User Models	95

2.6.2	Coupling Systems	96
2.6.3	State-Management Systems	98
3	Entity Taxonomy	102
3.1	Introduction	102
3.2	Application Domain vs. UI Domain State	103
3.3	Development of a Unified Model	107
3.4	A State Classification Based on Entity Properties	109
3.4.1	View Computation Function	114
3.4.2	Controller	114
3.4.3	Data Perspective	115
3.4.4	Timer Perspective	117
3.4.5	Mobile Model	117
3.4.6	Immutable Model	118
3.4.7	Immobile Model	118
3.5	Data Caching	119
3.6	Mobile Entity Migration	119
3.7	Entity Classification Summary	123
4	Concur Requirements and Architecture	125
4.1	Requirements	125
4.2	Elements of a Solution	126
4.2.1	The <i>Push</i> Model-View-Controller Paradigm	126
4.2.2	Logically Centralized Architecture	129
4.2.3	Common Hierarchical Data Modeling Facility	131

4.2.4	Continuously Evaluated Functional Views	136
4.2.5	Declarative User Interfaces	139
4.2.6	Perspectives	140
4.2.7	Composition Functions	151
4.3	Concur Architecture	153
4.4	Debugging, Testing, and Scripting	170
5	Concur Implementation	173
5.1	Programming Language and Libraries	173
5.2	Concur Class Library	176
5.3	Server Process	180
5.4	Client Process	180
6	Analysis and Evaluation of Concur	182
6.1	Criteria for Analysis	182
6.2	Experimental Environment	184
6.3	Experiments Performed	187
6.4	Entity Types and Applications	195
6.4.1	Text Editor Application	198
6.4.2	Pixel Editor Application	219
6.4.3	Jigsaw Puzzle Application	229
6.4.4	Application Summary	236
6.5	Determinism	236
6.6	Divergence and Modes of Work	238
6.7	Performance	239

7	Summary and Future Work	257
7.1	Summary	257
7.2	Future Work	258
7.2.1	Applying Concur to the Worldwide Web	258
7.2.2	Entity Taxonomy Maturity	264
7.2.3	Divergence	264
7.2.4	Continuously Evaluated Functions	265
A	Experiment Automation with the Puzzle Solver	267
A.1	Puzzle Solver Motivation	267
A.2	Puzzle Solver Overview	268
A.3	Puzzle Solver Algorithm	272
B	Exceptionally Long Latencies in the Puzzle Graph	279
	Bibliography	284

List of Tables

1.1	Basic Experiments	38
1.2	Experiment Dimensions	38
3.1	Entity Classes	113
4.1	Summary of Divergence Possibilities	144
4.2	Maximum Degrees Fahrenheit	156
6.1	Experiment Dimensions	194
6.2	Application Component Code Line Counts	197

List of Figures

1.1	A Taxonomy of Collaboration	3
1.2	Definitions	5
1.3	Multiple Model Scheme for Improving Performance	9
1.4	Simple Spreadsheet	11
1.5	Spreadsheet Presentations	12
1.6	The Model-View-Controller (MVC) Paradigm.	13
1.7	Spreadsheet MVC Organization	14
1.8	View/Controller Consolidation	14
1.9	Reused Model	15
1.10	Reused View	16
1.11	Pull MVC	16
1.12	Push MVC	17
1.13	Pull MVC Example	18
1.14	Push MVC Example	19
1.15	Cascading the Observer Design Pattern to Produce Layers	20
1.16	Example Centralized Architecture	22
1.17	Example Replicated Architecture	23
1.18	Detectable Inconsistency in a Replicated System	25
1.19	Model Synchronization in a Replicated System	26
1.20	Differing Projection Sequences Caused By Replicated Model Synchronization	27

1.21	Latency in a Centralized System	29
1.22	Multi-centered Systems with Entity Migration	34
1.23	Telegraphed User Intentions	35
1.24	Experimental Network	39
1.25	Centralized Architecture Latencies	44
1.26	Latencies by Architecture	45
1.27	Migration Latency over Time	46
1.28	Advantage of Prediction based on Telegraphed User Intentions	47
1.29	Advantage of Prediction based on Telegraphed User Intentions when Partial Migration is Included	48
1.30	Latency and Task Duration By User Count, Latency 0ms	49
1.31	Latency and Task Duration By User Count, Latency 50ms	50
1.32	Latency and Task Duration By User Count, Latency 100ms	51
1.33	Centralized and Migrating Message Counts	51
1.34	Message Counts by Architecture	52
1.35	Server CPU Utilization	53
1.36	Client CPU Utilization	54
1.37	Latency Distribution By User Count	55
1.38	Latency Distribution By User Count	56
1.39	Task Completion Times By Architecture	57
1.40	Migration Performance Under Contention	58
1.41	Migration Thrashing With and Without Prediction	59
2.1	XTV	61
2.2	XTV Latecomer Support	63

2.3	XTV Client Migration	64
2.4	Rendezvous	65
2.5	Weasel	67
2.6	Clock Architecture Example	69
2.7	Detail of Clock Example	70
2.8	ClockWorks Client/Server Annotation	72
2.9	Entity Classes	83
2.10	Entity Attributes	84
2.11	Centralized and Replicated Architectures	101
3.1	GMD Shared Object Model	104
3.2	GMD Coarse-Grained Sharing	105
3.3	Notepad	105
3.4	GMD Notebook UI State	106
3.5	GMD Conditional Data Structure	106
3.6	View and Model State Equality	107
3.7	Flattened State Organization	108
3.8	Coupled and Divergent State	108
3.9	Unified Model State	109
3.10	Dewan’s Generic Collaboration Architecture	110
3.11	Master/Slave Replication in a Centralized System	119
3.12	Master/Slave Migration in a Centralized System	120
3.13	Mobile Models	121
4.1	The Model-View-Controller Paradigm	127

4.2	Multiple Views of a Model	127
4.3	Hierarchical Data Structure	134
4.4	Graph implemented as Hierarchy with Special Links	134
4.5	Bubbling of Events or Changes	135
4.6	Same Model, Perspective, and View	145
4.7	Same Model and Perspective, Different View	145
4.8	Same Model and View, Different Perspectives	146
4.9	Same View and Perspective, Different Model	147
4.10	Same Model, Different Perspective and View	148
4.11	Same Perspective, Different Model and View	149
4.12	Same View, Different Model and Perspective	150
4.13	Different Model, Perspective, and View	151
4.14	Concur Architecture	154
4.15	Concur Architecture with Tags	155
4.16	Public Model State	158
4.17	View Specification	159
4.18	Projection	160
4.19	View Specification with Labels	161
4.20	Projection with Labels	161
4.21	Perspective	162
4.22	View Specification with Sliders	163
4.23	Projection with Sliders	163
4.24	View Specification with Changed Slider Values	164

4.25	Zoomed Projection	165
4.26	Controller Structure	166
4.27	Projection after Deleting a Point	167
4.28	Projection in Bar Chart Form	169
4.29	Software Architecture with a Scripting Layer	170
6.1	Background Network Traffic	186
6.2	Background Traffic Impact on Drag Latency (Centralized)	187
6.3	Background Traffic Impact on Drag Latency (Migrating)	188
6.4	Background Traffic Impact on Drag Latency (Replicated)	189
6.5	Puzzle Starting Point	189
6.6	Puzzle in Early Stage of Completion	190
6.7	Puzzle in Late Stage of Completion	190
6.8	Complete Puzzle	191
6.9	Text Editor Application	198
6.10	MetaModels.xml	199
6.11	TextEditorMetaModel1.xml	199
6.12	TextEditorMetaModel2.xml	200
6.13	ViewFunctions.xml	200
6.14	ControllerMaps.xml	201
6.15	Perspectives.xml	201
6.16	TextEditorCursor.xml	201
6.17	Applications.xml	202
6.18	TextEditorProducer.tcl	203

6.19	TextEditorModel.xml	203
6.20	TextEditorConsumer1.tcl	204
6.21	TextEditorViewFunction.xml Overview	205
6.22	TextEditorViewFunction Constructor	207
6.23	TextEditorViewFunction ViewFunctionInterestComplete	208
6.24	TextEditorViewFunction MetaModelInterestComplete	208
6.25	TextEditorViewFunction RequestParameter	209
6.26	TextEditorViewFunction ParameterReceived	209
6.27	TextEditorViewFunction ModelInterestComplete	210
6.28	TextEditorViewFunction ModelNotify_create	210
6.29	TextEditorViewFunction CreateLine	211
6.30	TextEditorViewFunction ModelTextChanged	211
6.31	TextEditorViewFunction SetCursor	211
6.32	TextEditorViewFunction CursorPositionChanged	212
6.33	TextEditorViewFunction ControllerMapInterestComplete	212
6.34	TextEditorControllerMap.xml Overview	214
6.35	TextEditorControllerMap Constructor	215
6.36	TextEditorControllerMap ViewSpecNotify_create	215
6.37	TextEditorControllerMap.xml CreateBindings	216
6.38	TextEditorControllerMap Key	217
6.39	TextEditorControllerMap ViewSpecTextChanged	218
6.40	TextEditorControllerMap Button-1	218
6.41	Pixel Editor Application	220

6.42	Pixel Editor Application Showing Area Boundaries	221
6.43	Pixel Editor MetaModel1	221
6.44	Pixel Editor Model	223
6.45	PixelEditorViewFunction.xml Overview	224
6.46	PixelEditorViewFunction ModelInterestComplete	224
6.47	PixelEditorViewFunction ModelNotify_create	225
6.48	PixelEditorViewFunction UpdateViewSpec	225
6.49	PixelEditorViewFunction ModelTileChanged	226
6.50	PixelEditorControllerMap Overview	227
6.51	PixelEditorControllerMap ButtonPress-1	228
6.52	PuzzleMetaModel1.xml	229
6.53	PuzzleProducer.tcl Overview	230
6.54	Puzzle Model	231
6.55	PuzzleViewFunction.xml Overview	233
6.56	PuzzleControllerMap.xml Overview	234
6.57	PuzzleFragmentPerspectiveManager.xml Overview	235
6.58	Overlapping Latencies while Dragging a Puzzle Piece	240
6.59	User Perceived Latency Distribution by Introduced Latencies	241
6.60	User Perceived Latency Distributions by Architecture	242
6.61	Migrating and Replicated Latencies (The Centralized plot is off the graph to the right.)	244
6.62	Latency Distribution by Background Traffic	245
6.63	Latency Distribution by User Count	246

6.64	Latency Distribution by User Count, Migrating and Replicated (The Centralized plot is off the graph to the right.)	247
6.65	Distribution of Work Picking Up Piece by Architecture - Initiator . .	249
6.66	Distribution of Work Picking Up Piece by Architecture - Other Clients	250
6.67	Distribution of Work Picking Up Piece by Architecture - Server . . .	251
6.68	Distribution of Work Dragging a Piece by Architecture	252
6.69	Distribution of Work Moving Cursor With and Without Prediction - Initiator	253
6.70	Distribution of Work Moving Cursor With and Without Prediction - Other Clients	254
6.71	Distribution of Work Moving Cursor With and Without Prediction - Server	255
7.1	The HTTP Protocol	259
7.2	HTTP Streaming	261
7.3	Nested Requests with Streaming	261
7.4	HTTP and Socket Connection	262
A.1	Color Quadrant Matching Computation	274
B.1	Long Latency Replay Tool in Action	280

List of Abbreviations

ALV	Abstraction-Link-View
CDF	Cumulative Distribution Function
CLOS	Common Lisp Object System
DAG	Directed Acyclic Graph
DOM	Document Object Model
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
LAN	Local Area Network
MVC	Model-View-Controller
MPVC	Model-Perspective-View-Controller
RVL	Relational View Language
UI	User Interface
UNC	University of North Carolina
URL	Uniform Resource Locator
WAN	Wide Area Network
WYSIWIS	What you see is what I see
WYSIWYG	What you see is what you get

List of Symbols

$\cong$		approximately equal
C		command causing state machine transition
I		input to a view function
M		model input to a view function
O		outputs emitted by a state machine
P		output of a projection function
p		projection function
π		perspective input to a view function
V		output of a view function
v		view function
$\mathcal{V}$		instantiation of a view function (with specific parameters)
$\mathcal{X}$		uncontrolled inputs to a state machine

Chapter 1

Introduction, Definitions, and Overview

1.1 Collaboration and Virtual Things

There has always been a need for people to work together, or collaborate, to do what individuals working alone cannot do. Before the invention of the Morse telegraph[Mor08] in 1837, communication across distances was too slow and unreliable to support most forms of collaboration. Thus, people generally had to be in the same location to work together. In modern times the telegraph, the telephone, and computer networks have made collaboration across distances routine.

Less obviously but just as significantly, we are now at the first point in history where many (perhaps most) of the things we wish to collaborate on and with are *virtual* rather than *real*, because many real things are being modeled by virtual counterparts on computers. Real things modeled by virtual counterparts can be as abstract as an idea or as concrete as a rock, as big as the universe or as small as a sub-atomic particle, as professional as a contract or as personal as a wink. Any real *thing*, using the term in its broadest sense, can be modeled by a virtual counterpart. Common examples are control panels, documents, books, file cabinets, calendars,

alarm clocks, money, maps, drawings, pictures, movie theaters, musical performances, games, shops, and business processes. Finally, virtual things can often be connected to their corresponding real things, providing the ability to monitor and manipulate things in the real world via virtual counterparts. Printers, scanners, cameras, microphones, and speakers provide simple examples, connecting virtual documents, pictures, and sounds to real ones. A more elaborate example is a bomb-disposal robot[Bom08] linked to a virtual (virtual reality) computer-modeled counterpart, the utility of which should be evident.

Virtual things can be given properties and behavior that their real counterparts cannot have. Virtual processes and objects can be slowed down or sped up, abstracted or magnified, simulated, modified, manipulated, tested, and observed in ways that may be expensive, difficult, unsafe, or impossible in the real world. And unlike physical things, virtual things can exist in more than one place at the same time. We can invert that concept and say that virtual things can reside at points in a virtual space orthogonal to real space. This enables real people to operate in the same virtual space, whether or not they are in physical proximity to each other.

Collaboration involves bringing people together in such a virtual space. If they are there at the same time such collaboration is called *synchronous*. If the space is persistent and people come and go independently, we term it *asynchronous* collaboration. If they are in near physical proximity to each other we say that they are colocated; if not, they are distributed. These two dimensions, time and physical place, define the simple taxonomy of collaborative systems shown in Figure 1.1, which was first proposed by Robert Johansen in [Joh88], and has been adopted by many others since.

Examples of collaborative tools in each of the four quadrants are as follows:

		Time	
		Same	Different
Place	Same	Synchronous Co-Located	Asynchronous Co-Located
	Different	Synchronous Distributed	Asynchronous Distributed

Figure 1.1: A Taxonomy of Collaboration

- **Synchronous Co-Located:** A projector used in a meeting room context facilitates communication among people meeting together at the same time and in the same place.
- **Asynchronous Co-Located:** A physical bulletin board is a means for communicating among people who pass through a given location at different times.
- **Synchronous Distributed:** The telephone is the most common technology supporting people who need to communicate at the same time from different locations.
- **Asynchronous Distributed:** Voice mail and email enable people to communicate while they are in different locations and cannot or prefer not to meet simultaneously.

1.2 Virtual Representations of Real Things

Before proceeding further, I will define a number of terms that will be useful in our discussion of how real things are represented as virtual things by a computer

(Figure 1.2). Unless otherwise noted, these definitions are my own and are tailored to the work presented in this dissertation, but the definitions presented here are similar to commonly-used definitions of these terms (i.e., their dictionary definitions).

A *representation* of one thing is another thing that “stands for” or “takes the place of” the first thing. This is the standard dictionary definition of “representation”. For example, my high school diploma is a representation of the fact that I graduated from high school, a picture of me represents my appearance, my watch represents the current time, and a map of the University of North Carolina (UNC) campus represents the layout of the streets and buildings on that campus. Representations are not unique. For example, “1”, “ e^0 ”, “one”, and “uno” all represent the abstract mathematical concept of the number one. Representations can also be ambiguous. For example, the name “John” currently refers to at least eight people in the Department of Computer Science at the University of North Carolina.

I will use the term *semantics* to refer to the abstract, essential *meaning* of a thing, independent of any representation of that thing. This is a slight generalization of the computer science use of the term *semantics* as defined in [Web06]¹. Often, the semantics is the referent of a representation. For example, the fact that I graduated from high school is the semantics behind my diploma, my appearance is the semantics behind my photograph, the abstract mathematical concept of the number one is the semantics behind the representations of *one* listed in the previous paragraph, and the actual UNC campus is the semantics behind a map of that campus. It is because the semantics are the same that we can say that “1”, “ e^0 ”, “one”, and “uno” all *mean the same thing*.

Semantics can be static or dynamic. The *static semantics* of a thing consists of its

¹From Webopedia: “In computer science, the term (semantics) is frequently used to differentiate the meaning of an instruction from its format.”

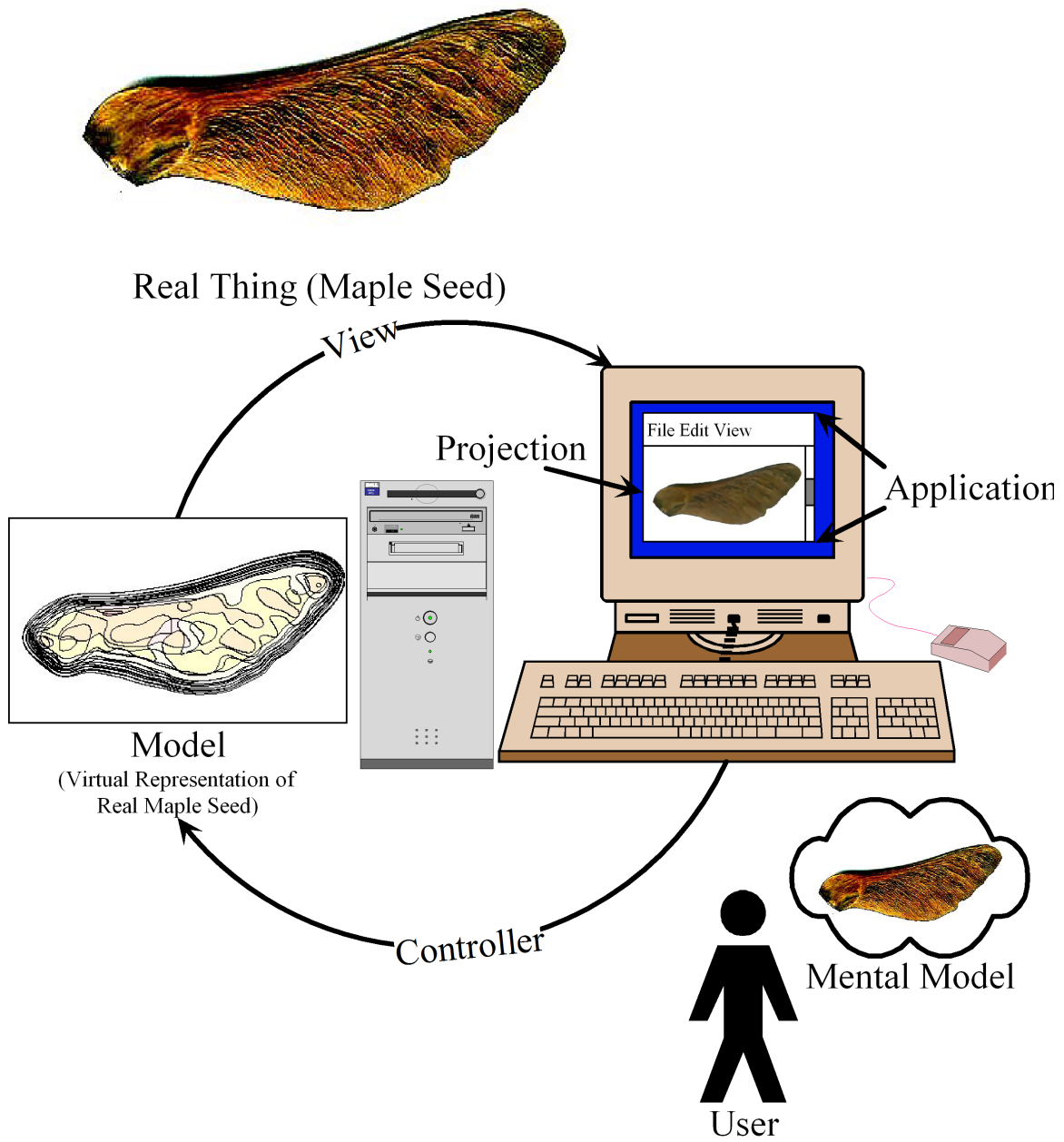


Figure 1.2: Definitions

abstract state at a point in time. Thus, the static semantics of my watch right now is the time 11:59am, and the static semantics of the local weather at this moment is “partly cloudy and warm”.

The static semantics of a thing can change over time. For example, the static semantics of my watch is now “12:00 noon”, and the static semantics of the weather at 5:00pm last evening was “cloudy and cool”. The *dynamic semantics* of a thing, or its *behavior*, is an abstraction of how the abstract state of a thing changes in response to internal or external stimuli. For example, my watch increments by one second in response to the internal stimulus of the passage of one second of time; this is the behavior of my watch. My alarm clock, however, sets its wake-up time in response to the external stimuli of “hour” and “minute” button presses; this is (part of) the behavior of my alarm clock.

A *model* is a software component that is a computer representation of a subset of the semantics (both static and dynamic) of a real thing. The purpose of a model is to represent a real thing in a computer-accessible and/or -manipulable form. For example, the current real time is represented as a model on my computer, in the form of a 32-bit integer that whose current value is the number of seconds since 12:00am GMT on January 1, 1970. By my definition, models are assumed to represent their referent semantics faithfully. That is, they are the authority (within the computer system) for the semantics of the referred-to thing. While, like any representation, they need not represent all aspects of their referent, they are not allowed to represent semantics that do not refer to their referent. That is, they are allowed to only represent the relevant *portions* of a real thing, but they may not *add* meaning that is not present in the real thing. The only exception to this rule is if the real thing is being represented virtually so that it can be “slowed down or sped up, abstracted or magnified, simulated, modified, manipulated, tested, and observed in ways that may

be expensive, difficult, unsafe, or impossible in the real world”, as described in Section 1.1. Even in these cases, any additional semantics are intended to reflect those of the corresponding real referent, but in ways that are inaccessible using the real referent. The point here is that if a model were to represent semantics that do not correspond to its referent, it would be modeling something other than (or in addition to) its referent.

A *projection* or *presentation* of a model (I use the terms interchangeably) is a user-accessible representation of the model, through which a user may sense and (optionally) interact with (invoke operations on) the model. A projection is usually a 2D representation of (some of) the externally-visible state and behavior (semantics) of the model. Often it is an approximation of a geometric projection onto a plane of the real object referred to by the model. Because they are concrete and not abstract (that is, they do more than simply represent the essential, abstract semantics of a thing), projections (unlike models) can present information in addition to what is present in their referent. This is the *syntactic sugar* referred to in [Dew99]. Syntactic sugar can include such things as decorations, labels, colors, and help text – everything, in fact, that is not a representation of the model semantics. A projection does not need to be a direct representation of a single model; it can be a representation of one model or of a combination of models, plus syntactic sugar. As with any representation, there are an infinite number of projections that can be produced from a single set of models at any point in time. Unlike models, projections are entirely derived from models and syntactic sugar, and are therefore non-authoritative with respect to the semantics of the underlying real things. That is, any software component desiring to access the semantics of the modeled real things must contact the model, not the projection, for those semantics.

A *view* is a software component that maps one or more models to a projection.

For the purposes of this dissertation, I will consider a view to ideally be a function (in the mathematical sense) whose domain is the externally-visible state of one or more models, and whose range is the set of resulting projection states. Since the underlying model is dynamic, the function is continuously evaluated so that it always produces a particular projection of the input models. A *controller* is a software component that maps low-level (e.g., keyboard and mouse) events applied to a projection into operations on the underlying models.

A user's *mental model* of a computer system is his understanding of the static and dynamic semantics underlying the projections through which he performs his work. Ideally, there is a strong correlation between the user's mental model and the real world being modeled by the computer system.

An *application* is simply an aggregation of projections. Thus, if I speak of the semantics of an application, I mean the aggregated semantics of the models underlying the set of projections of which the application is composed. For example, a typical text editor application models a control panel (using menus and buttons), a document, and document navigation facilities (using, e.g., scroll bars). The semantics of the text editor application are the aggregated semantics of the control panel, document, and navigation facilities.

1.3 Synchronous Distributed Collaboration

The primary focus of this work is on the synchronous distributed quadrant of Figure 1.1. Because of synchronicity, this is the quadrant where the performance demands are greatest. Because of distribution, this is also the quadrant where they are most difficult to achieve. Meeting these performance demands in the synchronous distributed quadrant ensures that they can also be met in the other three quadrants.

Typical work usually involves multiple collaborative quadrants and frequent transitions among them, as well as transitions to and from individual (non-collaborative) work. It is important for collaborative systems to support smooth (ideally seamless) transitions among these types of work, and I have endeavored to do so in the work described in this dissertation.

The difficulties encountered when attempting to provide reasonable performance in synchronous distributed collaborative systems have forced designers to make trade-offs in areas other than performance, compromising desirable characteristics in these other areas. For example, multiple, per-participant models of a single real thing are often used to increase interactive performance for each participant (Figure 1.3).



Figure 1.3: Multiple Model Scheme for Improving Performance

Unfortunately, this scheme invariably allows the models to diverge independently to some degree. Since participants in synchronous distributed collaboration are working together closely, these differences are likely to confound collaborative work by either making the existence of differing models visible to participants (due to the tight

synchronization of their work), or by presenting inconsistent views of the world to different participants undetectably (due to the distribution of participants). Either way, these are undesirable properties of virtual things as compared to the real things they are intended to represent. Real things are unique and therefore fully consistent with themselves.

Another undesirable characteristic of multiple-model schemes is that they are much more difficult to design and implement than single-model schemes. Keeping multiple models synchronized is complicated, particularly because they must reside in different (distributed and non-identical) environments. Shielding participants from differences in the models or making such differences understandable to participants also presents enormous difficulties not encountered in the single-model case. Finally, multiple-model schemes complicate transitions to other quadrants of Figure 1.1 and to individual work, where multiple model schemes are generally unnecessary.

This dissertation focuses on the task of providing good interactive performance to all participants of synchronous distributed collaborative work while retaining simple, consistent user mental models and implementations, and seamless transitions among modes of work.

1.4 Separation of Models and Views

A widely used engineering principle for interactive software applications is to cleanly separate models from views. As an extremely simple example, consider the three-cell spreadsheet of Figure 1.4. There are various ways to define the model for even this simple application. I will use the following definitions:

- The *state* of the model consists of the numbers in cells A1, A2, and A3.

	A
1	1 4 7 3 7 6 . 4 1
2	6 5 0 4 7 . 1 4
3	2 1 2 4 2 3 . 5 5

Figure 1.4: Simple Spreadsheet

- The *behavior* of the model is that cells A1 and A2 can be modified to contain any numbers, and cell A3 always contains their sum.

This model is independent of any particular presentation. In addition to the presentation shown in Figure 1.4, any of the presentations of Figure 1.5 may be used. The set of all possible presentations of this or any model is infinite.

It is apparent from Figure 1.5 that the *state* of an application can be presented in different ways. What is not apparent from this figure, but is nonetheless part of model/view separation, is that the *behavior* of the application can be presented in different ways. For example, one might be able to modify the values of the top two cells by editing the numbers in the upper textual presentations of the figure, or by dragging lines in the lower graphical presentations.

Clean separation of the semantic and presentation aspects of an application is particularly important for synchronous collaboration, because such collaboration involves multiple participants viewing and interacting with a shared application model. Presentation independence makes it easy to attach multiple views to a shared model. It also facilitates the provision of different presentations to each participant, a likely scenario given that different participants may have different preferences or may fulfill different roles.

	A
1	147,376.41
2	+65,047.14
3	212,423.55

Stocks	\$147,376
Bonds	\$65,047
Total	\$212,424

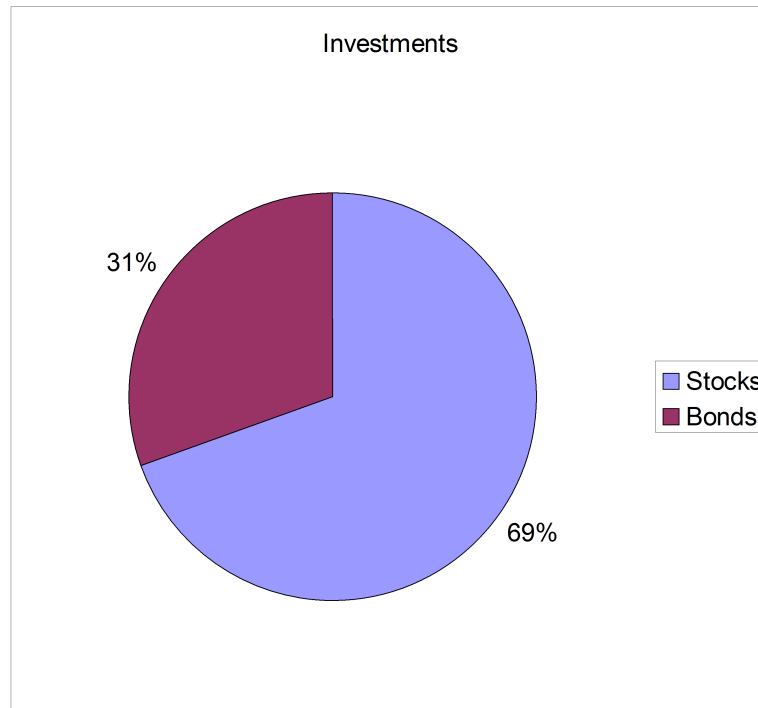
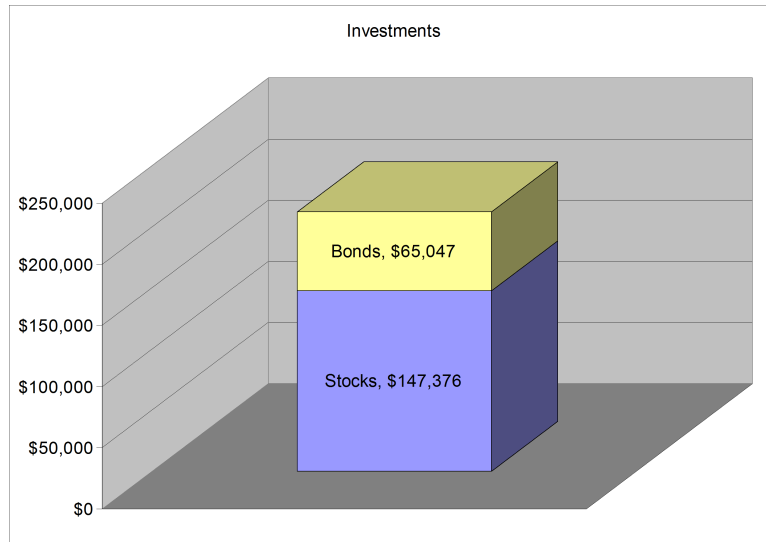


Figure 1.5: Spreadsheet Presentations

1.5 The Model-View-Controller Paradigm and the Observer Design Pattern

In order to provide clean separation of semantics and presentations, interactive software applications and infrastructures commonly use some variant of the Model-View-Controller (MVC) paradigm illustrated on the left side of Figure 1.6. In this

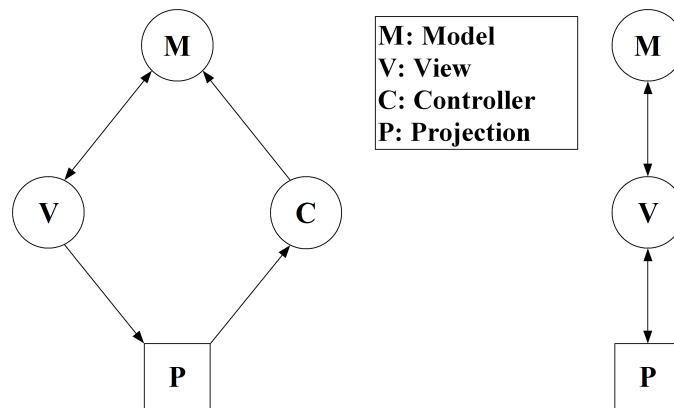


Figure 1.6: The Model-View-Controller (MVC) Paradigm.

figure, circles represent software components, arrows represent method calls or events, and squares represent the product of views (projections). In this paradigm the model *M* maintains semantic state and behavior, the view *V* maintains a projection of the model, and the controller *C* transforms user inputs applied to the projection into operations on the model. The pseudo-code of Figure 1.7 sketches an MVC organization of the application of Figure 1.4. Because the view and controller are intimately tied to the user interface (projection) and to each other, and because interactions (bindings of keyboard and mouse events to operations on a model) are rarely customized separately from the view, controller functions are often subsumed by the view as demonstrated by the right side of Figure 1.6 and the pseudo-code of Figure 1.8. The primary advantage of MVC is that it separates models and views so that they can be

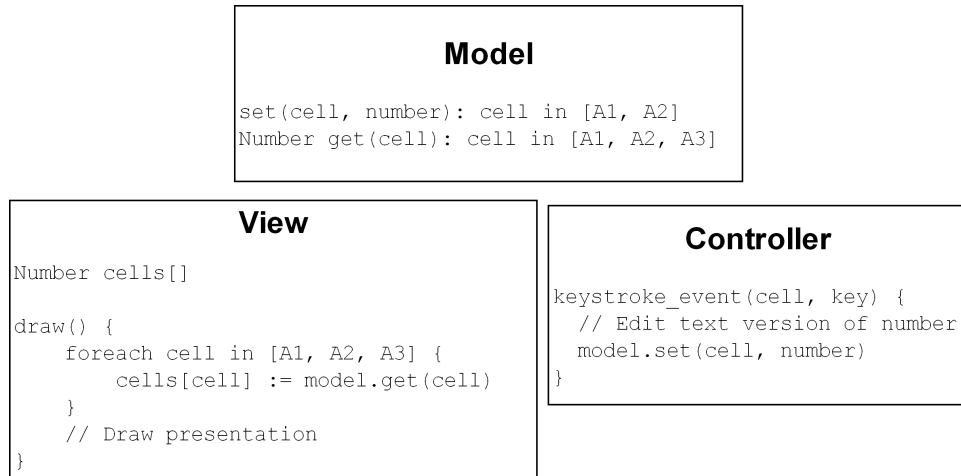


Figure 1.7: Spreadsheet MVC Organization

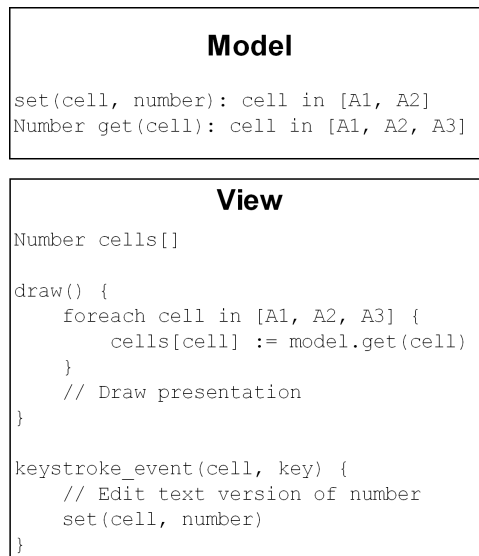


Figure 1.8: View/Controller Consolidation

reused with other views (Figure 1.9) and models (Figure 1.10), respectively.

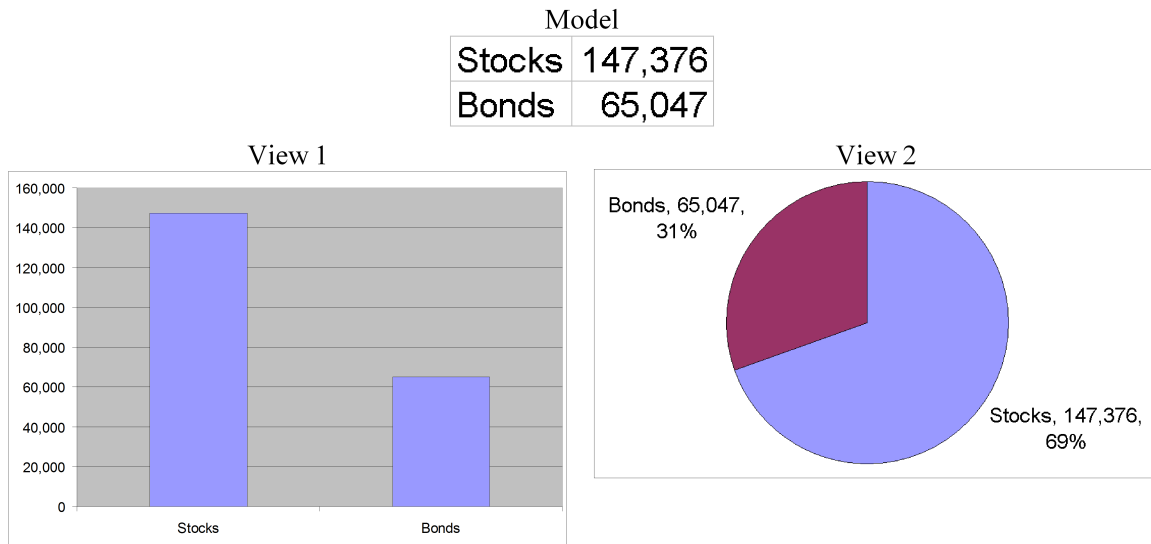


Figure 1.9: Reused Model

Two important variants of the MVC paradigm are *pull* (Figure 1.11) and *push* (Figure 1.12) MVC². In the pull variant the view pulls data from the model, while in the push variant the model pushes data to the view. The pull variant is the classical MVC paradigm proposed in [KP88] and utilized by Smalltalk. The model need know nothing about the view except that it is interested in changes in the model. The view needs to know more about the model in order to query it for its current state, but it can be reused with other models supporting the same interface. Figure 1.7 has been fleshed out as Figure 1.13 to demonstrate pull MVC.

The push variant is more efficient than the pull variant because the model only notifies a view of changes in the particular aspects of the model in which the view is interested, and because the model sends descriptions of the changes with the notification. The trade-off is that this requires more knowledge of the view by the model.

²Pull and push MVC are described in more depth in the Observer design pattern [GHJV95a], which will be introduced shortly.

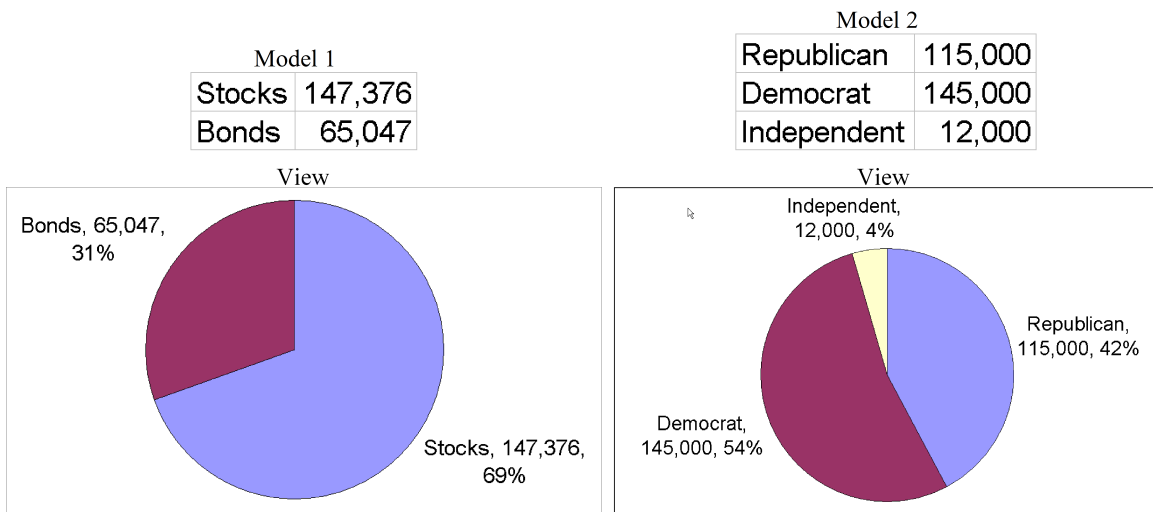


Figure 1.10: Reused View

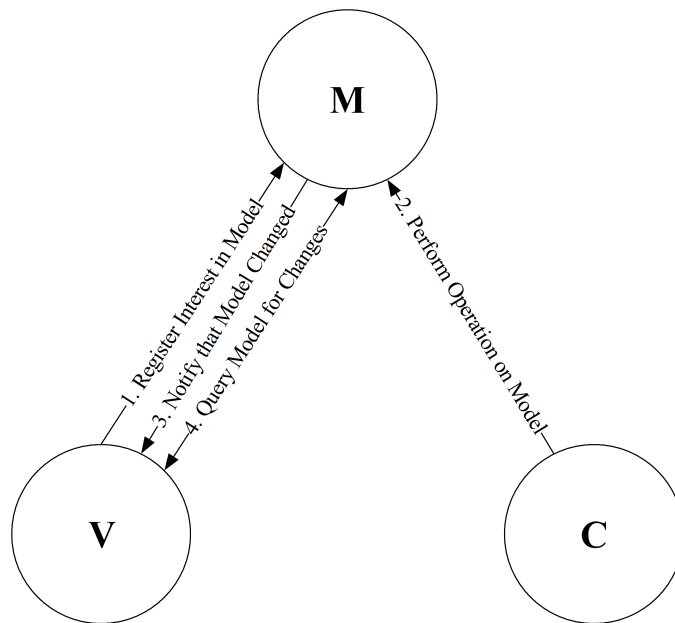


Figure 1.11: Pull MVC

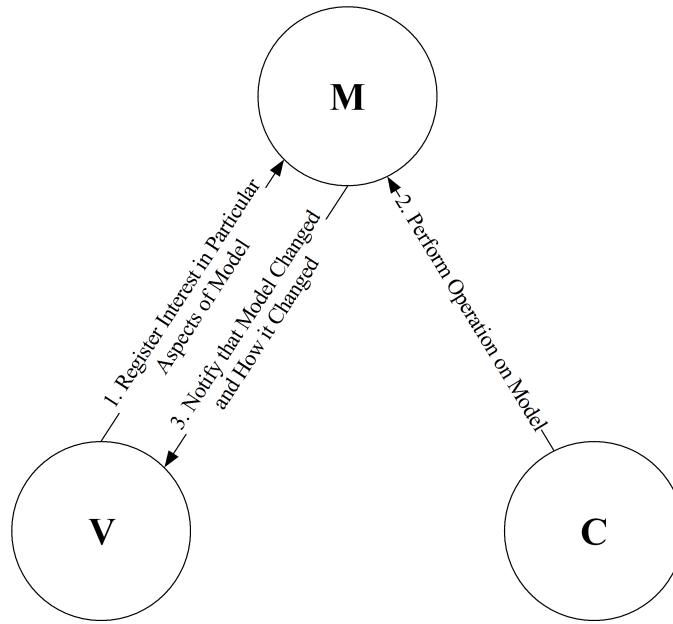


Figure 1.12: Push MVC

That is, the model must have more specific information about the view's interests, and it must match model changes to these interests. To demonstrate the push variant, we need a slightly more elaborate application with specific interest specifications by views. Figure 1.14 sketches an example application where the model is a list of stocks and their current quotes, while the view selectively displays stock symbols and quotes. Note that the model only notifies views that have expressed an interest in the particular stock whose quote has changed. Interests can be more complex than this example demonstrates. For example, a view might only want to be notified when stock S reaches a quote of at least Q . Note that in a pull implementation of this application the model would notify all views when any stock changes, and each view would have to pull the quotes of all stocks in its interest set to determine which, if any, had changed, and whether the change was significant to the view.

A more recent generalization of the Model-View separation technique is found in

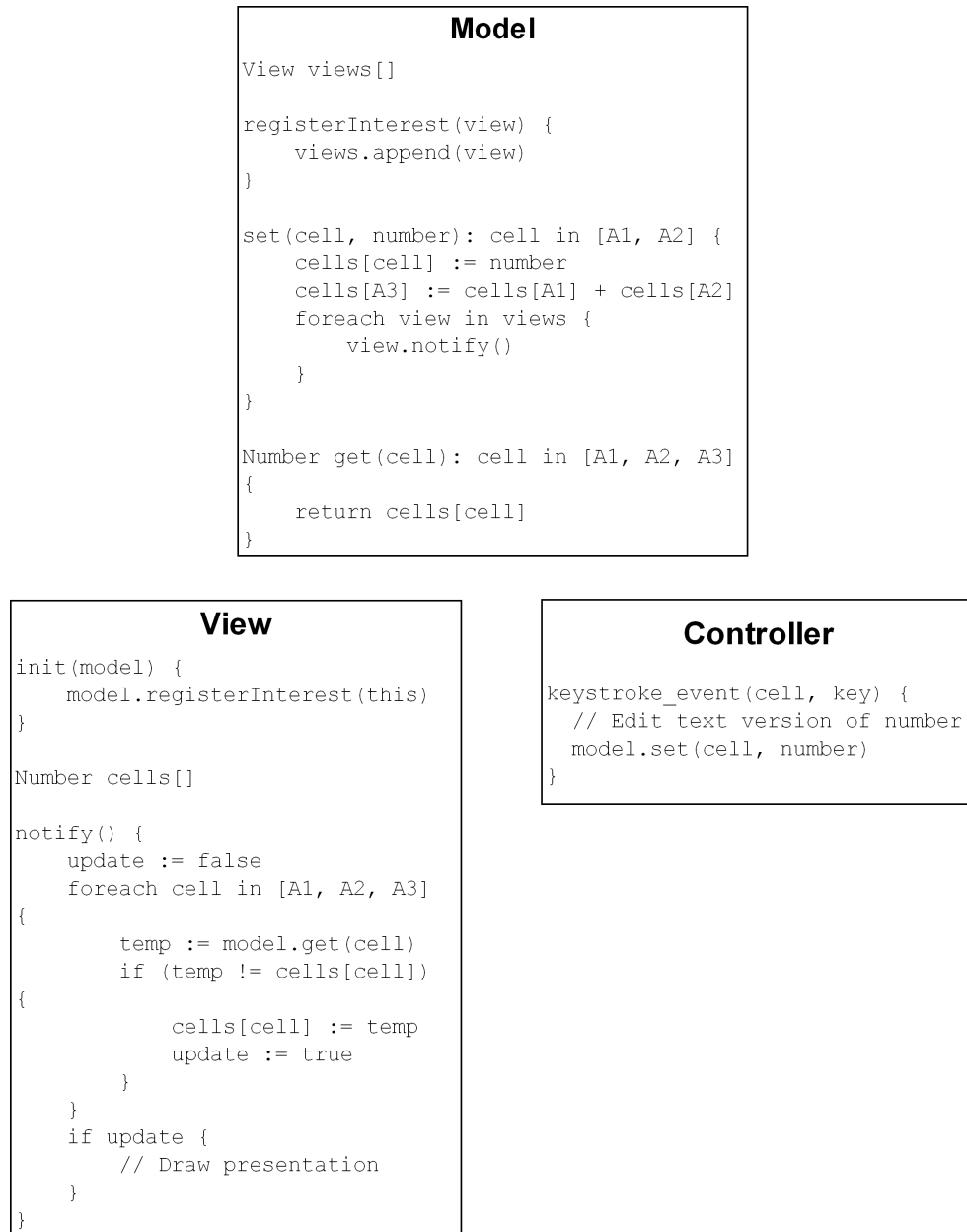


Figure 1.13: Pull MVC Example

Model

```
View views[][]

registerInterest(view, stock) {
    views[stock].append(view)
}

set(stock, quote) {
    cells[cell] := quote
    foreach view in views[stock] {
        view.notify(stock, quote)
    }
}
```

View

```
init(model) {
    foreach stock in [HPQ, MSFT] {
        model.registerInterest(this, stock)
    }
}

notify(stock, quote) {
    // update presentation
}
```

Figure 1.14: Push MVC Example

the *Observer* design pattern [GHJV95a]. A *design pattern* is a generic object-oriented solution to a software engineering problem that shows up in many contexts³. The Observer design pattern uses the terms *Subject* for Model and *Observer* for View, where the observer can be any object interested in the subject's state changes, not just *view* user interface elements. The Observer design pattern may be cascaded to produce multiple layers, as shown in Figure 1.15. The middle object in this diagram has two roles: it is an observer of the subject above and a subject for the observer below. Cascading can continue indefinitely, but is subject to per-layer performance overhead.

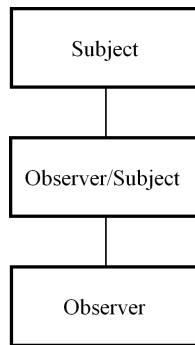


Figure 1.15: Cascading the Observer Design Pattern to Produce Layers

1.6 Centralized and Replicated Architectures

MVC and the Observer design pattern are applicable to a wide range of software systems, whether individual or collaborative, colocated or distributed, synchronous or asynchronous. Synchronous distributed collaboration obviously requires distribution of software and hardware components and data among multiple sites. Such

³MVC is also a design pattern. See [GHJV95b] for a complete discussion of these and other design patterns.

distributed systems require architects to make decisions about how components and computations are to be distributed.

Synchronous distributed collaborative systems have historically been categorized as having either centralized or replicated architectures. Different definitions of *centralized* vs. *replicated* architectures are used in the literature. (See, for example, [Dew99] and [GR99].) I will create and use the following definitions. My definitions generalize upon the definitions in [GR99], which are specified at a coarse application level of abstraction.

A *centralized* architecture (Figure 1.16) is one where each aspect of semantic state and its behavior is represented at any given time by exactly one master model.

Consider, for example, the application of Figure 1.16. All stocks could be represented by one model as shown, each stock could be represented by a separate model, or anything in-between. A given stock or all stocks could shift from one model instantiation to another. However, a given stock cannot be simultaneously represented by more than one model. There may be slave models corresponding to each master model (e.g., transparent observer/subjects used to increase performance or fault tolerance), but these are restricted to precisely tracking the master model's external state and its transitions.

By contrast,

A *replicated* architecture (Figure 1.17) is one where each aspect of semantic state and its behavior is represented simultaneously by multiple (peer) master models. These models are synchronized in some fashion but are not required to go through the same sequence of external state transitions.

That is, Model 0 $\cong$ Model 1 $\cong$ Model 2 in Figure 1.17. All three models represent the same stocks, stock *LEH* is the same in all three models, the quotes are all the same,

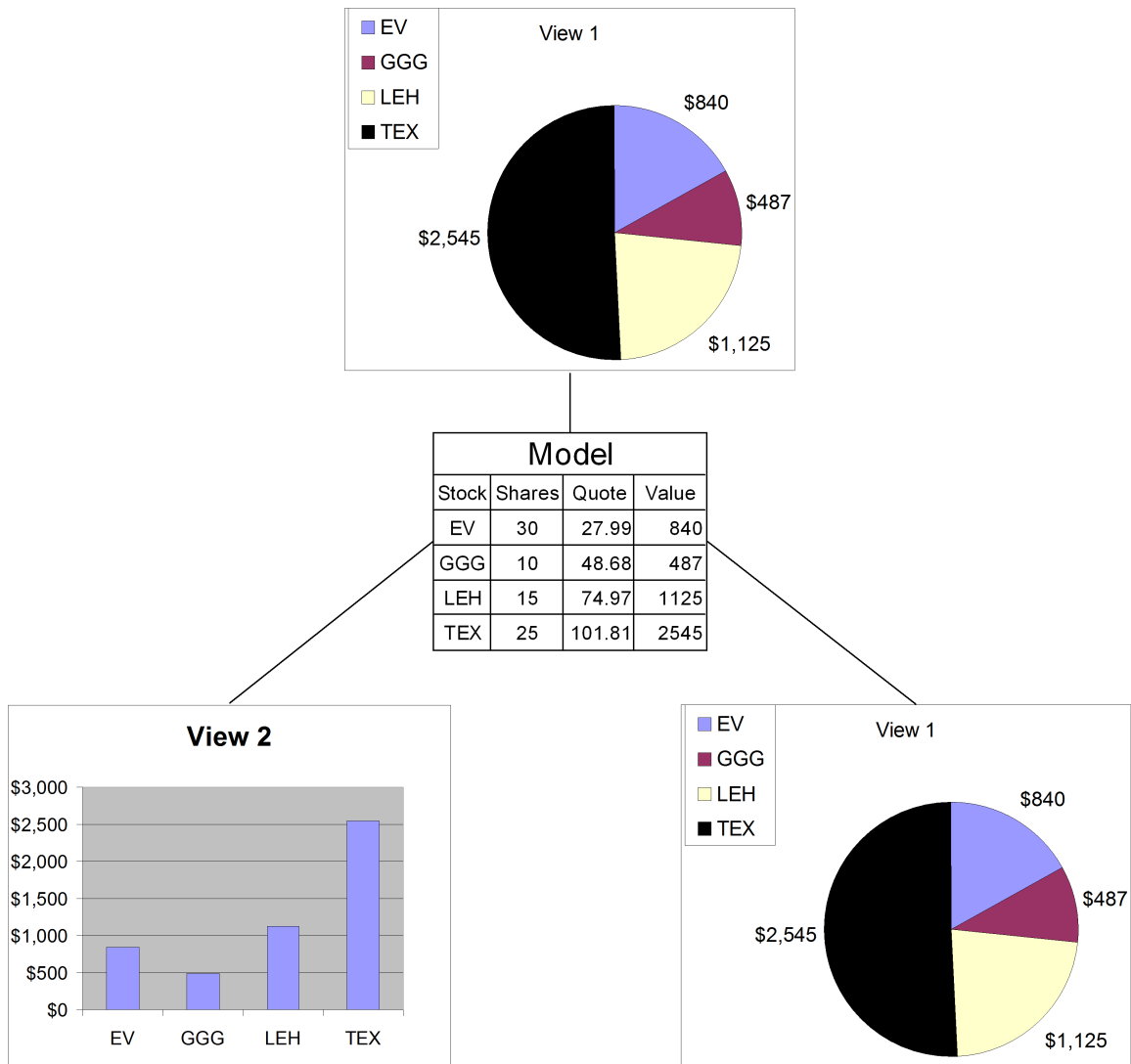


Figure 1.16: Example Centralized Architecture

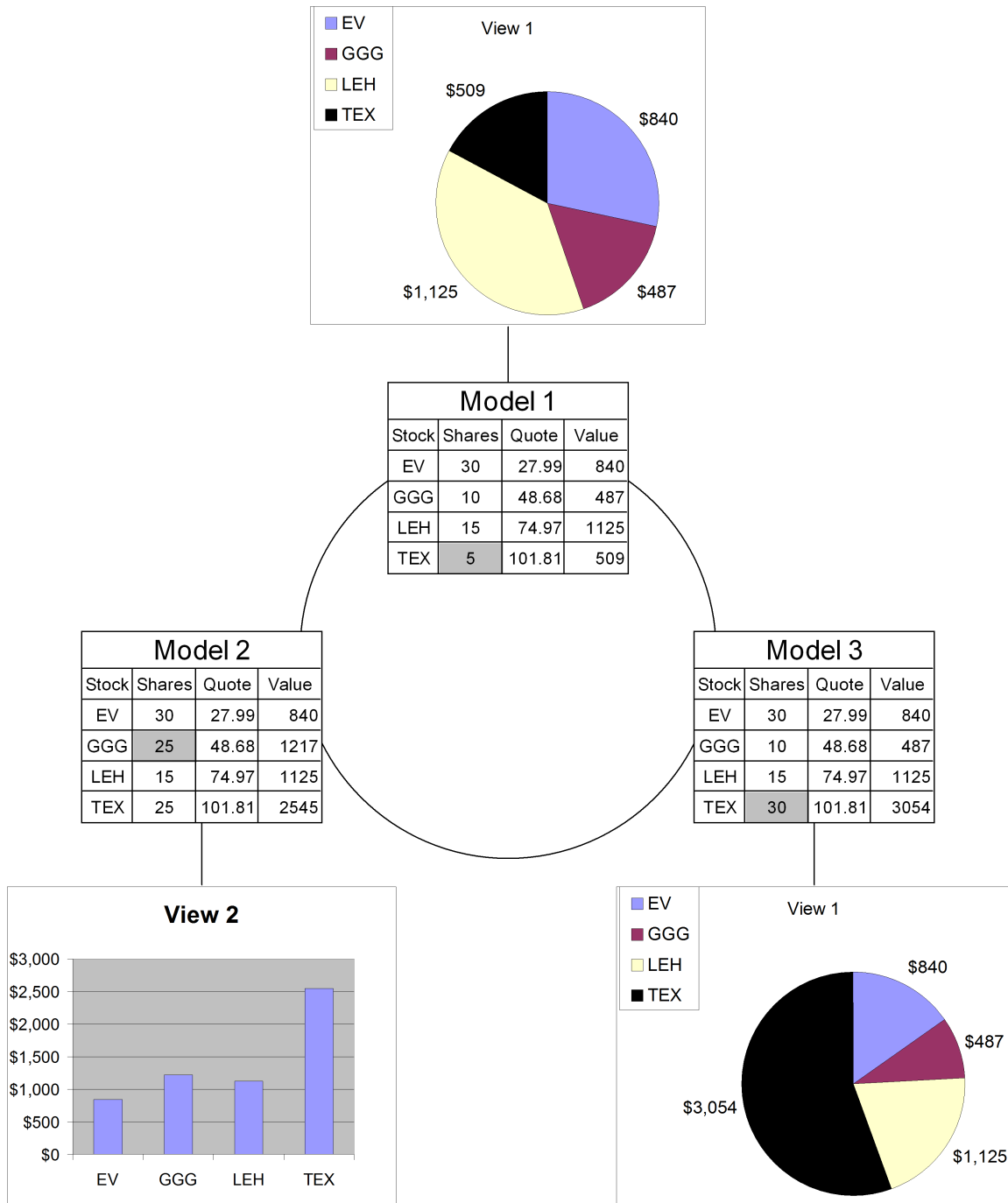


Figure 1.17: Example Replicated Architecture

and the values are all Shares times Quotes. However, the number of shares of the *TEX* stock differs among all three models and the number of shares of *GGG* in Model 2 differs from the number of shares in the other models. The fact that a given stock can simultaneously have multiple static semantics exposes the fact that a replicated architecture is being used; that is, that a given real thing is being represented by multiple models simultaneously.

Both centralized and replicated synchronous distributed collaborative systems have been built, each class having its characteristic advantages and disadvantages. Centralized systems support simpler user mental models and are easier to implement, but they suffer from greater latency and reduced scalability. Replicated systems improve latency and scalability at the cost of more complex user mental models and implementations.

To understand these trade-offs, consider again Figures 1.16 and 1.17. The user's mental model of the application as implemented by the centralized system of Figure 1.16 is simple, because he and his collaborators always see consistent, though perhaps different, projections of the model (ignoring different transmission times that will cause users' displays to be updated at slightly different times). Even considering network transmission times, all projections go through states corresponding to exactly the same sequence of model transitions. By contrast, users of the replicated system of Figure 1.17 must construct a more complex mental model of the application, because their projections need not conform to each other at any particular point in time, and the sequence of updates to one user's projection need not be consistent with the sequence of updates to another user's projection. Figures 1.18 and 1.19 demonstrate these problems. In Figure 1.18, the users may become confused because some out of band communication mechanism (e.g., audio or video) exposes the inconsistencies in model values. Furthermore, if each user changes the number of shares in his local

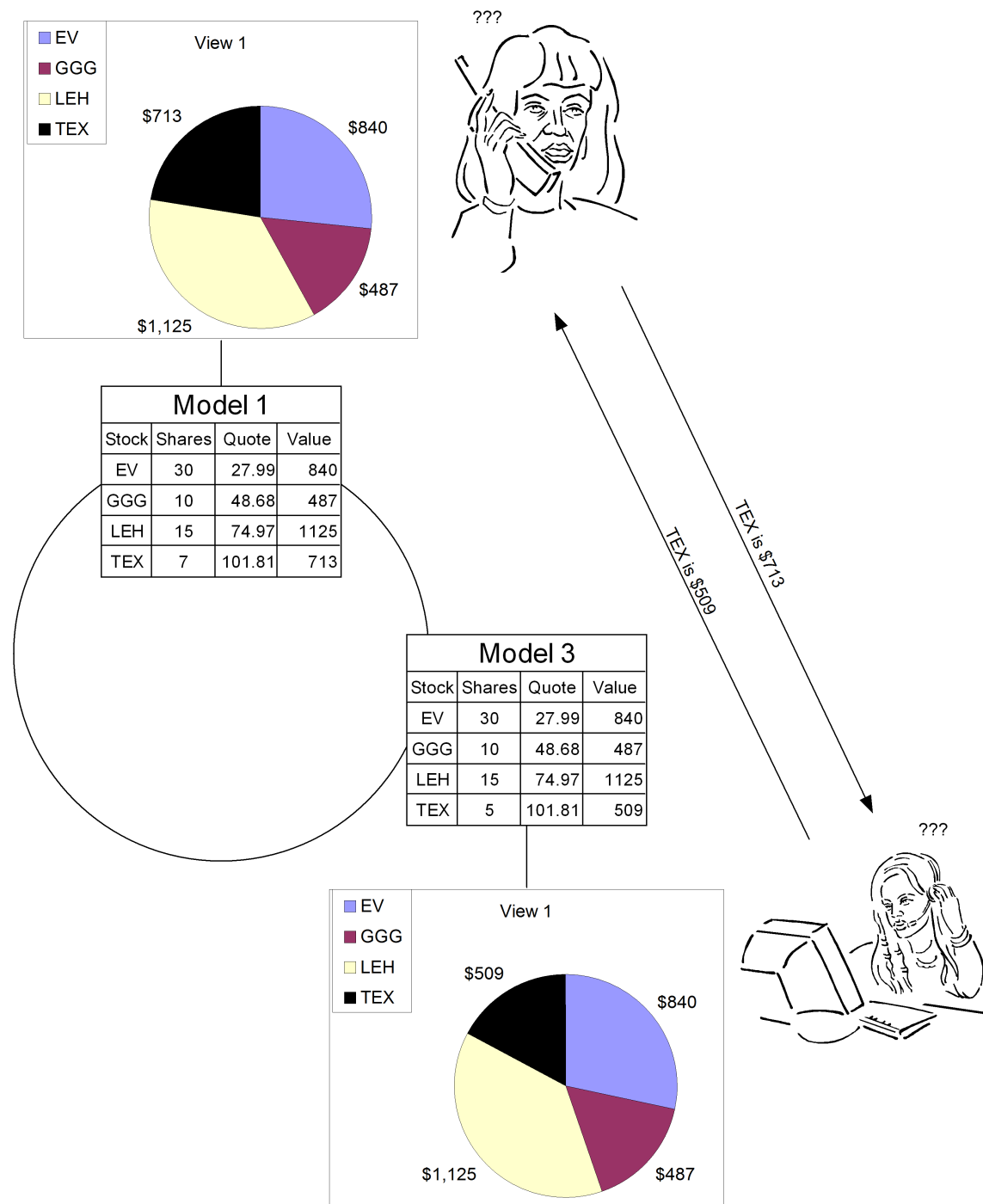


Figure 1.18: Detectable Inconsistency in a Replicated System

model copy as indicated by the shaded boxes of Figure 1.17, and the models are ultimately synchronized, the sequence of model (and therefore projection) changes may be inconsistent from user to user, as shown in Figures 1.19 and 1.20. Note that each

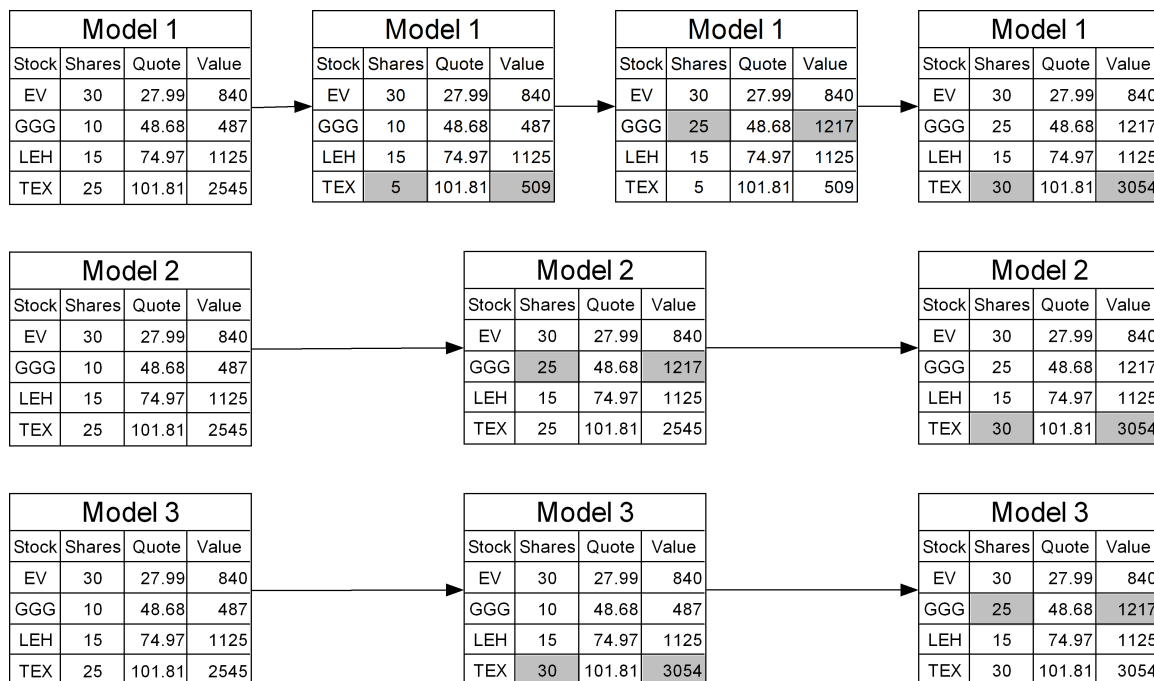


Figure 1.19: Model Synchronization in a Replicated System

participant starts and ends with the same projection, but the intervening projection sequences differ. (For clarity, I have given each user the same pie-chart projection and have only shown the pies themselves in Figure 1.20.) Whether or not these update sequence differences are of any consequence depends on the application. The point here is that the separate models change in ways that are inconsistent with viewing them as a virtual representation of a single shared real thing. Finally, it should be apparent that replicated systems are more difficult to implement, since they, unlike centralized systems, require synchronization of models.

On the other hand, replicated systems can typically provide much better interac-

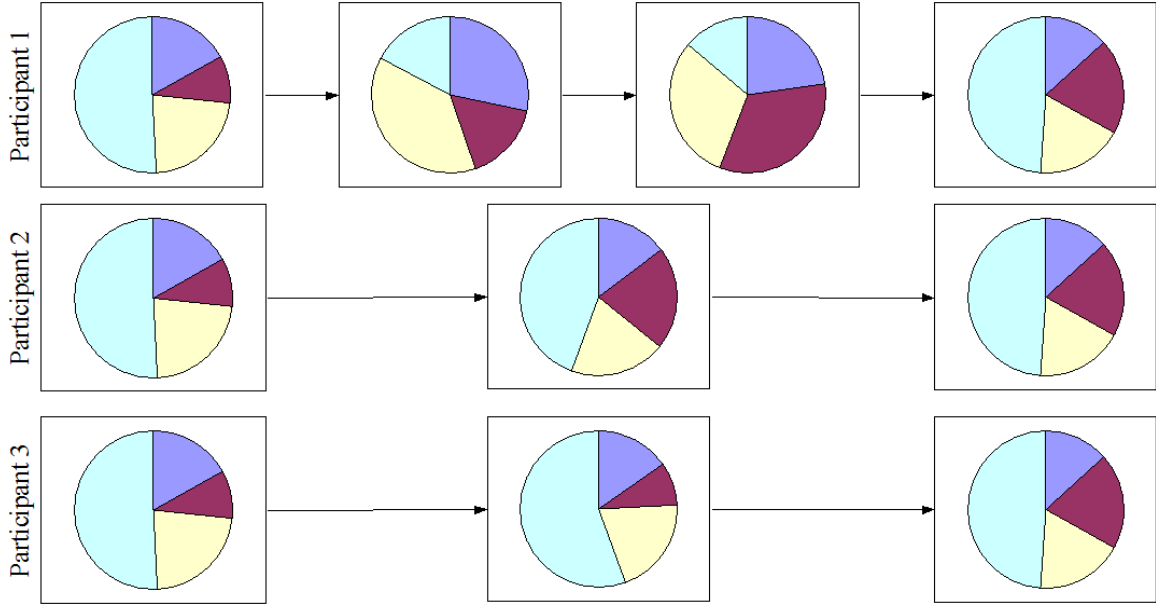


Figure 1.20: Differing Projection Sequences Caused By Replicated Model Synchronization

tive performance than centralized systems. There are three reasons for this:

- The impact of network latencies on interactive performance is much greater for centralized systems. For example, if network transmission time is 100ms, as shown in the centralized system of Figure 1.21, every user interaction (e.g., every incremental movement of the cards being dragged in the figure) will have a latency (i.e., the time between the user’s movement of the mouse and the corresponding movement of the cards) of at least 200ms. In a replicated system, the user dragging the cards would get immediate feedback.
- In a centralized system, the “center” of the system where the model resides can become overloaded if many participants join a collaborative session. In a replicated system this will not happen, because there is no unique “center” that must process all interactions. Thus, replicated systems are more scalable in terms of the computations that must be performed.

- A centralized system typically places more demands on network bandwidth than a replicated system, because lower-level operations are transmitted over the network (e.g., mouse movements instead of final card positions). Thus, like the processor at the center, the network is more likely to become overloaded. This, like the former point, means that replicated systems generally scale better than centralized ones.

The focus of this work is on scenarios involving highly synchronized collaboration, where multiple simultaneous participants react to other participants' actions on virtual things by invoking their own actions, all during a short (e.g., sub-second) time span. Examples are users working together closely on a game, a puzzle, or a project. Conceptually, the simpler user mental models supported by centralized architectures are better suited to these scenarios than the more complex mental models required by replicated architectures. The whole point of highly synchronous collaboration is to share some virtual thing at the current point in time. This implies that the model defining the real thing should be unique. In replicated architectures the approximations in the equation $\text{Model } 0 \cong \text{Model } 1 \cong \text{Model } 2$ can cause confusion for users, particularly if there are out-of-band communication channels that can redundantly communicate model state, as we saw in Figure 1.18. Replicated architectures are used where performance is at a premium and can be purchased at the cost of complexity for implementors and participants.

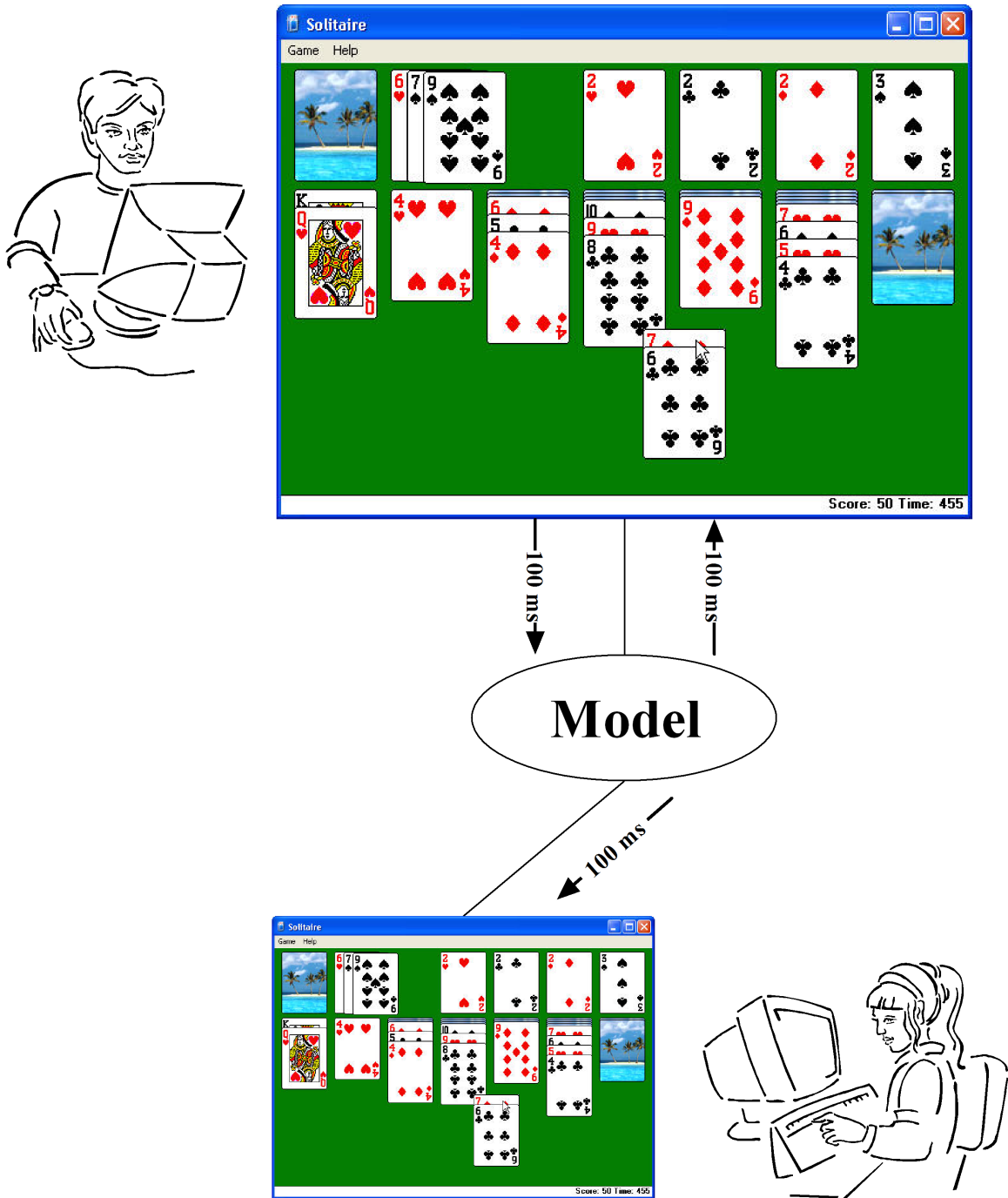


Figure 1.21: Latency in a Centralized System

1.7 Hybrid Architectures and Dynamic Reconfiguration

Attempts to resolve the conflicting characteristics of centralized and replicated architectures have resulted in hybrid architectures that are partly centralized and partly replicated (e.g., [RG97]). These systems apply centralized and replicated sub-architectures selectively in order to best fit their architectural properties to various aspects of the system. More recent research (e.g., [CD01]) has investigated support for dynamic reconfiguration among centralized, replicated, and hybrid configurations in response to the changing properties of a collaborative session over time. Dynamic reconfiguration typically involves migration of processes or objects among physically dispersed processors.

Unfortunately, hybrid and dynamic architectures further complicate the user's mental model, because they expose to the user more of the internal structure of the system and how it changes dynamically. Process and (object-oriented) object migration are also expensive in terms of migration time and the container requirements at various processors, where a *container* is the surrounding runtime environment that must be available to host a process or object. Process and object container requirements also tend to be extensive and operating-system and/or language specific. Finally hybrid and dynamic systems add another layer of implementation complexity to the infrastructure, and perhaps the applications.

1.8 Problem Statement and Thesis

Collaborative systems enable people to work together. They can be categorized along one dimension as either synchronous or asynchronous, and along another dimen-

sion as either colocated or distributed, as shown in Figure 1.1. Synchronous systems demand higher performance than asynchronous systems because they imply a high degree of interaction by participants. That is, participants must respond to actions by other participants within a short (e.g., sub-second) time-frame. It is more difficult to meet performance demands in distributed systems than in colocated systems, because distribution implies networks spanning larger distances, which means greater latencies and lower bandwidths than are typical for colocated systems. Synchronous distributed systems therefore have the highest performance demands and the most difficult environment for meeting those demands. If performance demands can be met in this quadrant of Figure 1.1, they can be met in all the quadrants.

Centralized, replicated, and hybrid architectures have been used to build synchronous distributed collaborative systems. Centralized systems suffer from long latencies because all model operations must go through the physical center of the system. They also scale poorly because they place more demands on the central computer and the network than do replicated systems. Replicated systems solve these performance problems by using distributed model replicas at each participant site. Unfortunately, these replicas are difficult to keep synchronized. Worse, replicated architectures necessarily allow the replicated models to diverge independently. Consequently, users either become aware that there are multiple divergent model replicas (complicating the user's mental model of the application), or they are presented with divergent versions of a world that is ostensibly the same, without the participants being aware that this is happening. This latter case can lead to inconsistent understandings of and actions on the virtual world by different participants. In either case, collaborative work is confounded by the replicated architecture.

Hybrid architectures attempt to apply centralized and replicated sub-architectures selectively to a single system in order to ameliorate these problems. Unfortunately,

this further complicates the programmers' and users' mental models because they have to make distinctions in their mental models between centralized and replicated aspects of the system. More recent research has focused on dynamically modifying the architecture of the system to meet the changing demands of a collaborative session over time. While this can improve performance and scalability, the implications for the user's mental model of the application are even worse than for a static hybrid system, since the user's mental model must adapt over time to the changing architecture. Dynamic architecture adaptation also typically requires mechanisms for migrating processes or objects. Process and object migration systems have extensive container requirements for distributed computers, and process and object migrations are slow as compared to user interaction times.

In this dissertation, I have taken a finer-grained approach to migration. Instead of migrating processes or even object-oriented objects, I have identified smaller and simpler *entities* that do not need to conform to full process or object semantics and that therefore have reduced container requirements. For example, an entity may be a simple data element represented in an open, standard format (e.g., string). As such, it does not require its own execution environment (thread) or data hiding facilities. I have defined and classified entities based on their migration characteristics. In so doing, I have identified a number of important entity types with low container requirements that can be migrated extremely quickly and easily.

The central thesis of this dissertation is that:

A taxonomy of application *entities* (objects with less than process and object-oriented object semantic requirements) based on their migration characteristics facilitates the construction of collaborative applications and supporting infrastructures that retain the simplified user mental model and implementation characteristics of centralized systems while achieving the enhanced latency and scalability characteristics of replicated systems.

This is achieved through the fast migration of lightweight entities in a multi-centered centralized system. I define a *multi-centered* system (Figure 1.22) as having a single physical center and many logical, per-entity centers that migrate independently and dynamically in response to user interactions. The speed of such lightweight entity migration opens up the possibility of triggering migrations based on telegraphed user intentions (user actions that hint at imminent succeeding actions), which are likely to be more accurate predictors of future interactions than are longer-term interaction histories. For example, in Figure 1.23, the user is telegraphing his intention to move the two of clubs by moving his cursor into an invisible “halo” around that card (made visible in the figure as a light-colored box). Relatively slow user interaction times can be overlapped with object migration, so that the objects being manipulated will have been migrated before they are manipulated (or, at worst, the migration will have begun). Identifying sub-object, easily-migratable entity classifications also minimizes container requirements, facilitating widespread entity distribution. I anticipate that this will help us to achieve the critical mass required for the acceptance of any communications system.

This dissertation also asserts and demonstrates that an entity classification based on migration characteristics serves well to identify and independently assign the entities users may wish to diverge upon in a collaborative session. That is, the entities one might wish to migrate for performance reasons are typically the very entities that one might want to diverge upon. This presents fine-grained, user-understandable, and application-unaware divergence possibilities along the full spectrum from WYSIWIS⁴ collaboration to independent work, and supports dynamic transitions along this spectrum. This capability is important because the desired degree of coupling in

⁴What You See is What I See. That is, everyone sees exactly the same thing.

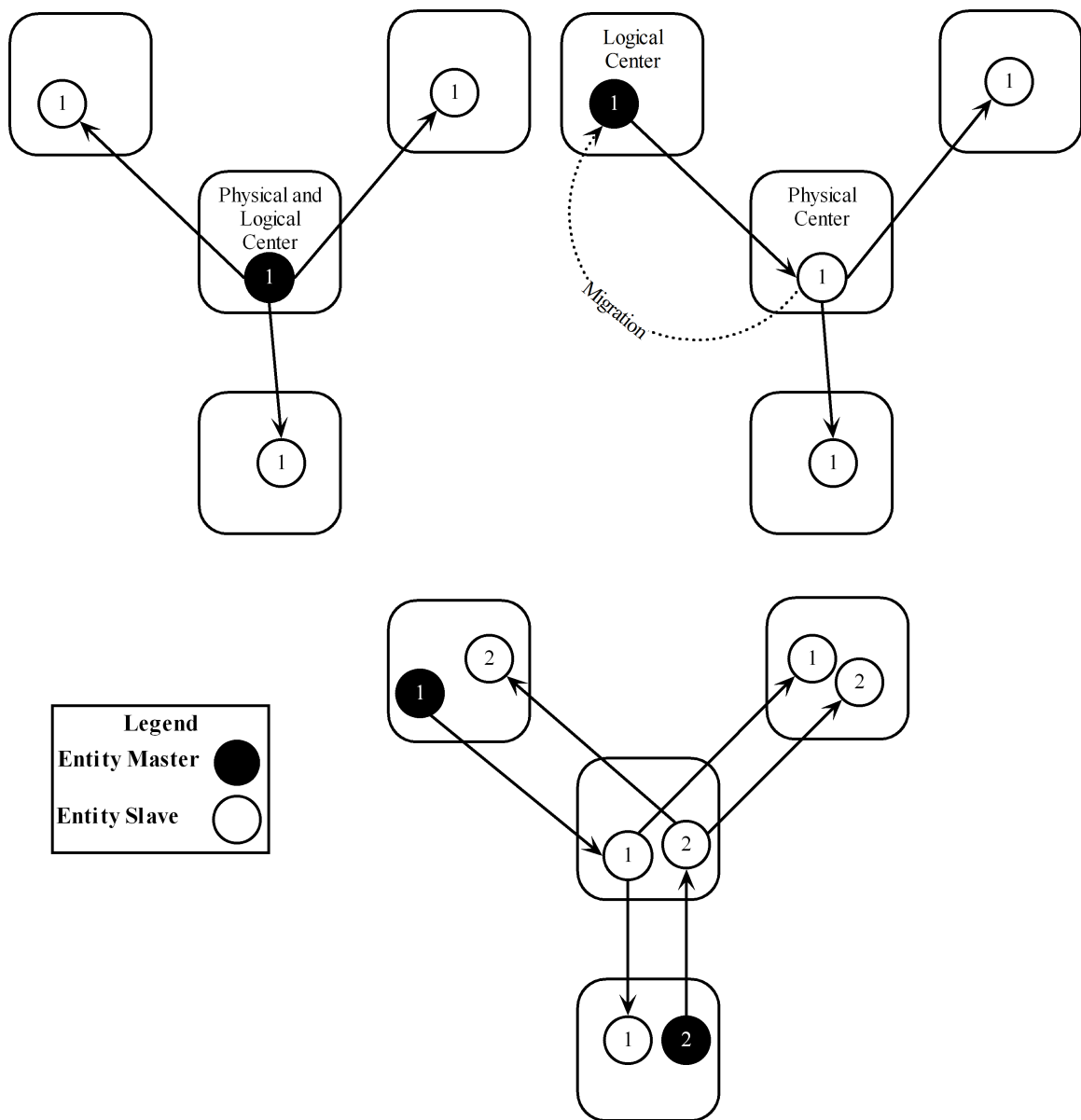


Figure 1.22: Multi-centered Systems with Entity Migration

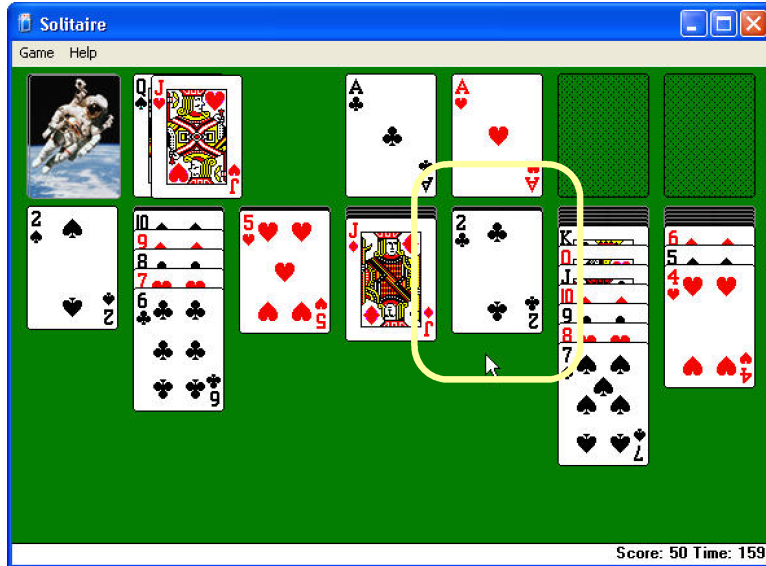


Figure 1.23: Telegraphed User Intentions

collaborative work has been shown both to vary depending on the task at hand, and to fluctuate during particular collaborative sessions.

1.9 Contributions of this Work

The following are the contributions of the work described in this dissertation:

- A novel classification of application software components (entities) based on their migration characteristics. This classification identifies a number of entity types that can be migrated very easily and quickly (as compared to user interaction times of 50-100ms[Shn98]). Experimental evidence is presented supporting migration speed claims.
- A demonstration that these entity types can be used to build a wide range of applications, in the form of implemented applications of representative types. These applications are written assuming a centralized infrastructure. The rep-

representative applications are a jigsaw puzzle, a text editor, and a pixel-based drawing editor. These were chosen because they require radically different ways of representing their model data (as objects, text, and pixels, respectively), and because they stress the infrastructure’s ability to provide both collaboration and local, high performance interaction with different interaction styles (using drag-and-drop, typing, and pixel painting, respectively).

- An informal demonstration that this infrastructure provides a reasonable and understandable programming environment based on a centralized architecture, in the form of sample code for the above applications. These applications are collaboration-unaware; that is, they do not need to be aware of the number or identity of participants.
- A prototype infrastructure supporting these applications, built as a multi-centered centralized system with entity migration support.
- Experimental evidence supporting my claim that this infrastructure and applications conforming to it give substantially better interactive performance to all participants than does a purely centralized architecture, and close to that of a replicated architecture.
- Experimental evidence supporting my claim that this architecture scales better in terms of both processor and network bandwidth utilization than a purely centralized architecture.
- An informal proof that this architecture does not require model synchronization algorithms akin to those used in replicated architectures, which simplifies the programming of the infrastructure and/or applications.

- An informal proof that the kinds user mental model complexities required by a replicated architecture are not present in this architecture.
- A demonstration of how the above-mentioned entities can be used to implement a wide range of desirable per-participant divergence scenarios.
- An analysis of predictive migration based on telegraphed user intentions rather than a past history of interaction.

1.10 Evaluation Summary

As my proof of concept, I have developed *Concur*⁵, a novel architecture and prototype implementation of an infrastructure supporting lightweight entity migration. Chapter 6 details the evaluation of this work with respect to performance (both latency and scalability), ease of application construction, and usability by participants. A summary of this evaluation is given here.

Evaluation for the performance criteria has been done formally via experiments on a laboratory network at the University of North Carolina (UNC) Department of Computer Science. All experiments were automated, with user interaction characteristics that model recorded traces of actual user interactions. Evaluation for the other two criteria (ease of application construction and usability) has been done less formally via demonstration applications and argument.

⁵**con·cur** \kən-'kər, kən-\ *vi* **con·curred**; **con·cur·ring** [ME *concurrēn*, fr. L *concurrere*, fr. *com-* + *currere* to run – more at CAR](15c) **1** : to act together to a common end or single effect **2 a** : APPROVE <~ in a statement> **b** : to express agreement <~ with an opinion> **3 obs** : to come together : MEET **4** : to happen together : COINCIDE *syn* see AGREE (WEBSTER'S Ninth New Collegiate Dictionary)

Architecture	Prediction
Centralized	N/A
Replicated	N/A
Migrating	no
Migrating	yes

Table 1.1: Basic Experiments

Dimension	Values
Network Delay Introduced (each way)	0 ms, 50 ms, 100 ms
UNC Traffic	no, yes
Users	1, 2, 4, 6

Table 1.2: Experiment Dimensions

1.10.1 Experiment Setup

The formal experiments (Tables 1.1 and 1.2) consisted of a jigsaw puzzle solved by varying numbers of users (clients), with three different architectures: centralized, replicated, and migrating. The tests with the migrating architecture were performed both with and without using a prediction mechanism for migrating puzzle pieces based on telegraphed user intentions. The puzzle application was chosen because it exhibits a range of collaboration, from mostly-independent work near the beginning of a given puzzle solution to mostly-collaborative work as the puzzle nears completion.

A single server computer hosted two processes: the Producer, which created and maintained the jigsaw puzzle model, and the Server, which served this model to the clients. Each client computer hosted a process containing the view computation function, whose output was projected locally via the X Window System. The same process also contained a special controller embodying the puzzle solving logic. The puzzle solver took into account the general shapes of the pieces (straight, male, or female edges) and a rough notion of the color of the half of the piece nearest an edge

to be matched, in order to simulate how a person would solve the puzzle. The puzzle to be solved was a picture of Franklin Street in Chapel Hill, NC.

The laboratory network (Figure 1.24) consisted of two gigabit Ethernet switches connected by routers. Each experiment was performed using artificially-introduced

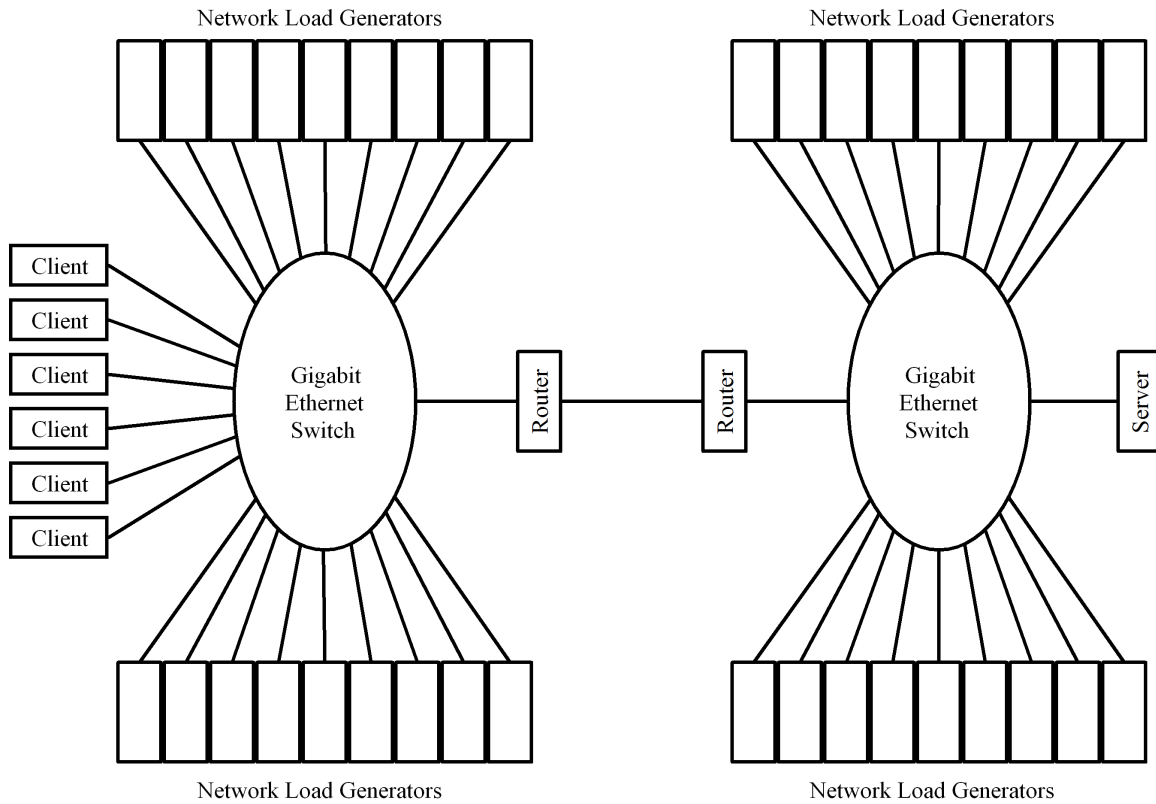


Figure 1.24: Experimental Network

network delays of 0, 50, and 100ms. Each was performed with 1, 2, 4, and 6 users solving the puzzle. Each was performed both with and without additional network load. The network load introduced was an actual reproduced load of the UNC network, from a trace taken in August of 2004. (See [HC06] for a description of how the capture and replay of this network load were performed.) This introduced traffic load had no significant impact on any of the experiments, so it will not be further discussed in this chapter. The graphs in this chapter were all computed from experi-

ments without UNC traffic.

Each experiment was repeated four times. The total number of experiment repetitions was 4 (basic experiments) $\times 3$ (network delays) $\times 2$ (with and without introduced network traffic) $\times 4$ (user counts) $\times 4$ (repetitions of each of these) $= 384$. Each of these 384 experiments was started with a unique arrangement of puzzle pieces randomly placed on the table (such that they could overlap). Puzzle pieces were not rotated (i.e., they always had the orientation they would have in the final picture), as rotation was deemed an unnecessary complication for the purposes of this experiment.

1.10.2 Notes on Experimental Result Plots in this Dissertation

Plots of experimental results in this dissertation (Figure 1.25, for example) show either a distribution of values recorded over the duration of an experiment (if the **Distribution** tab at the top is selected) or values recorded over time as the experiment progresses (if the **Value Over Time** tab is selected). Distributions are typically first plotted as a histogram of values recorded (where the X axis shows the values recorded and the Y axis shows the count of each value during the experiment). They are then plotted as a Continuous Distribution Function (CDF)[CDF08], where the X axis is the same as in the histogram, but the Y axis shows the percentile of the X value. That is, the Y value corresponding to an X value in a CDF shows the fraction of the total area under the histogram curve that is to the left of the X value.

The data from the experiments ultimately comes from log files that trace various events during experiment execution. A relational database was populated from these traces, organizing the data as required for plotting and analysis. The experimental result plots shown in this dissertation are created by an application that queries the

database using parameters selected from some of the drop-downs at the bottom of the graph, and plots it according to the values selected by other drop-downs. The drop-downs are defined as follows:

- **Architecture, Latency, Prediction, Traffic, and UserCount** - Selects one point along a dimension (e.g., the Centralized architecture), or a comparison of all points along a dimension. Comparisons can be plotted on a single graph or on separate graphs arranged vertically or horizontally.
- **Repetition** - Selects one repetition of the experiments, a comparison of all repetitions, or an average of the values over all repetitions.
- **Data** - Selects the data to be plotted.
- **CDF** - Determines whether distribution data should be plotted as a histogram (N) or CDF (Y).
- **Client** - Selects one client, a comparison of all clients, or an average of values over all clients.
- **Interval (Seconds)** - On **Value over Time** plots, selects the interval over which data is averaged. This determines the spacing of labeled data points on the graph.
- **Bucket Size** - Determines the bucket size for **Distribution** graphs.
- **X Max, Y Max** - Determines the maximum X or Y value to be plotted. Two drop-downs are used. The leftmost selects the significant digits and the rightmost selects the power of 10.
- **Y Min 0** - Should the Y axis be anchored at zero? (Y or N)

- **Smoothing** - Determines the type of smoothing to be done between data points (e.g., linear, or quadratic spline).
- **Legend** - Determines where to place the legend.

If a graph contains more than one plot, there is a legend identifying each plot by color and symbol. The following abbreviations identify the dimensions that differ among plots:

- **A** - Architecture
 - **C** - Centralized
 - **M** - Migrating
 - **R** - Replicated
- **L** - Introduced latency in milliseconds
- **P** - Prediction algorithm based on telegraphed user intentions used? (Y or N)
- **T** - Background traffic generated? (Y or N)
- **U** - User count - 1, 2, 4, or 6
- **R** - Repetition - 1, 2, 3, or 4
- **C** - Client - 1, 2, 3, 4, 5, or 6

1.10.3 Experimental Results

The primary measurement in my experiments was latency. Latency was measured during puzzle piece drag operations. It was calculated as the elapsed time from the user's initiation of a cursor movement while holding a piece, until the first draw

operation corresponding to that user action was executed. Each measured latency corresponded to moving the piece a few pixels, not to the entire drag of the piece. Thus, there were many measurements for one drag of a piece. See Section 6.7, and Figure 6.58 in particular, for more details.

The introduced network delays simulated the delays of a Local Area Network (LAN) (0 ms), intranet (50 ms), and Wide Area Network (WAN) (100 ms). Latency in a pure centralized system was demonstrated, as expected, to be $O(2 * \text{network transmission time})$ (Figure 1.25). Latencies with the migrating architecture were confirmed to be similar to those of the replicated architecture (Figure 1.26). This is the single most important result of my experiments.

Entity migration was confirmed to be $O(4 * \text{network transmission time})$, as expected (Figure 1.27). (Look at the data near the middle of each plot, after most pieces have been moved from the server to a client, and before contention for pieces begins to take over.)

Predictive migration techniques based on telegraphed user intentions were compared with prediction based on past interaction history. The prediction algorithm used was *cursor vectoring* (migrating puzzle pieces toward which the cursor is moving). Prediction based on past interaction history was implemented by simply leaving each piece where it was last used. Figures 1.28 and 1.29 show the advantage of using telegraphed user intentions as a prediction mechanism. The migration hit ratio is the probability that a piece is already migrated to the user's client when he picks it up to move it. (The former graph includes only completely migrated pieces, while the latter includes pieces that have begun but not completed migration.) The advantage decreases as the number of users increases, due to contention over pieces. This type of prediction is enabled by the speed of Concur's lightweight entity migration.

Scalability results were less dramatic, but still significant. Figures 1.30, 1.31, and

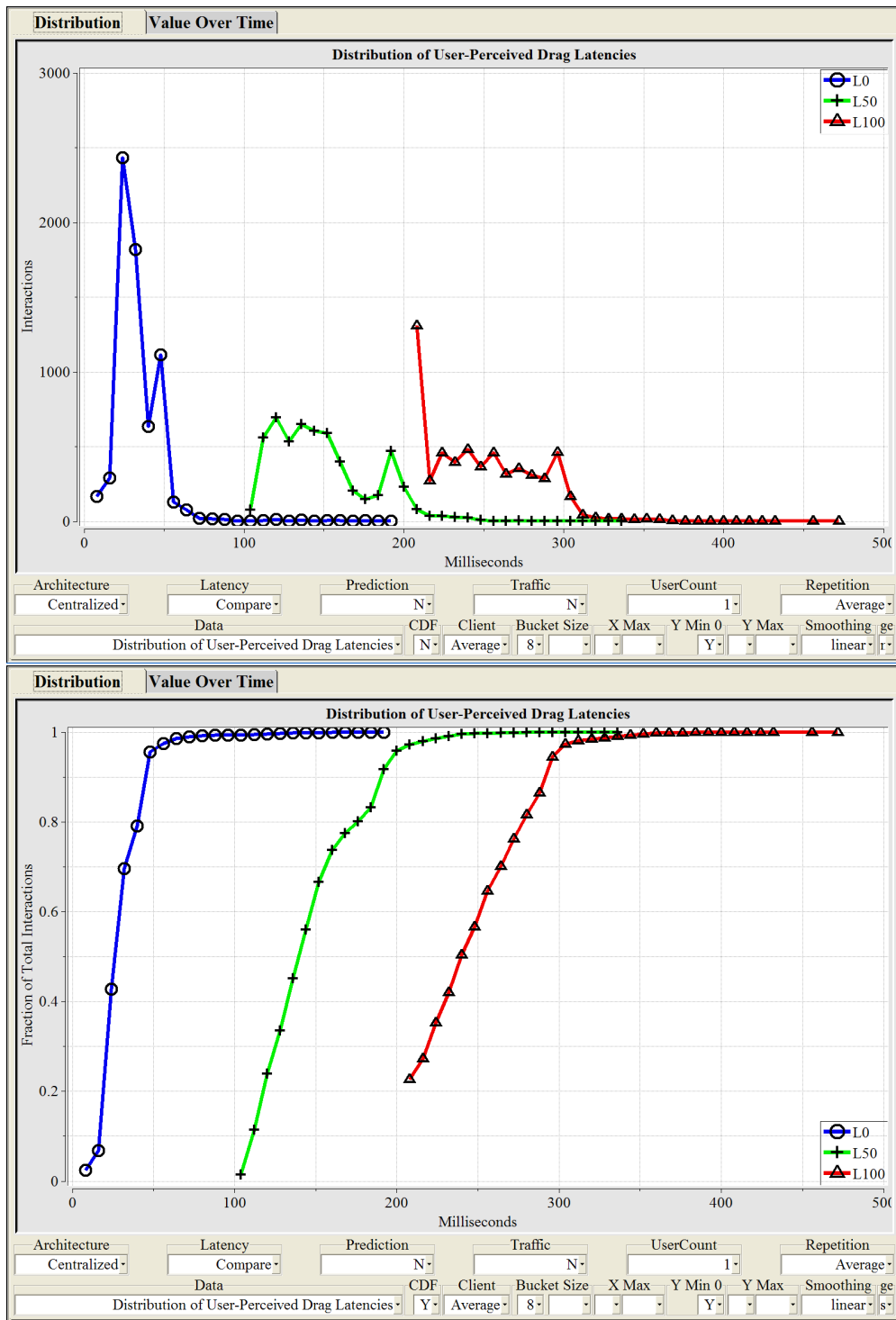


Figure 1.25: Centralized Architecture Latencies

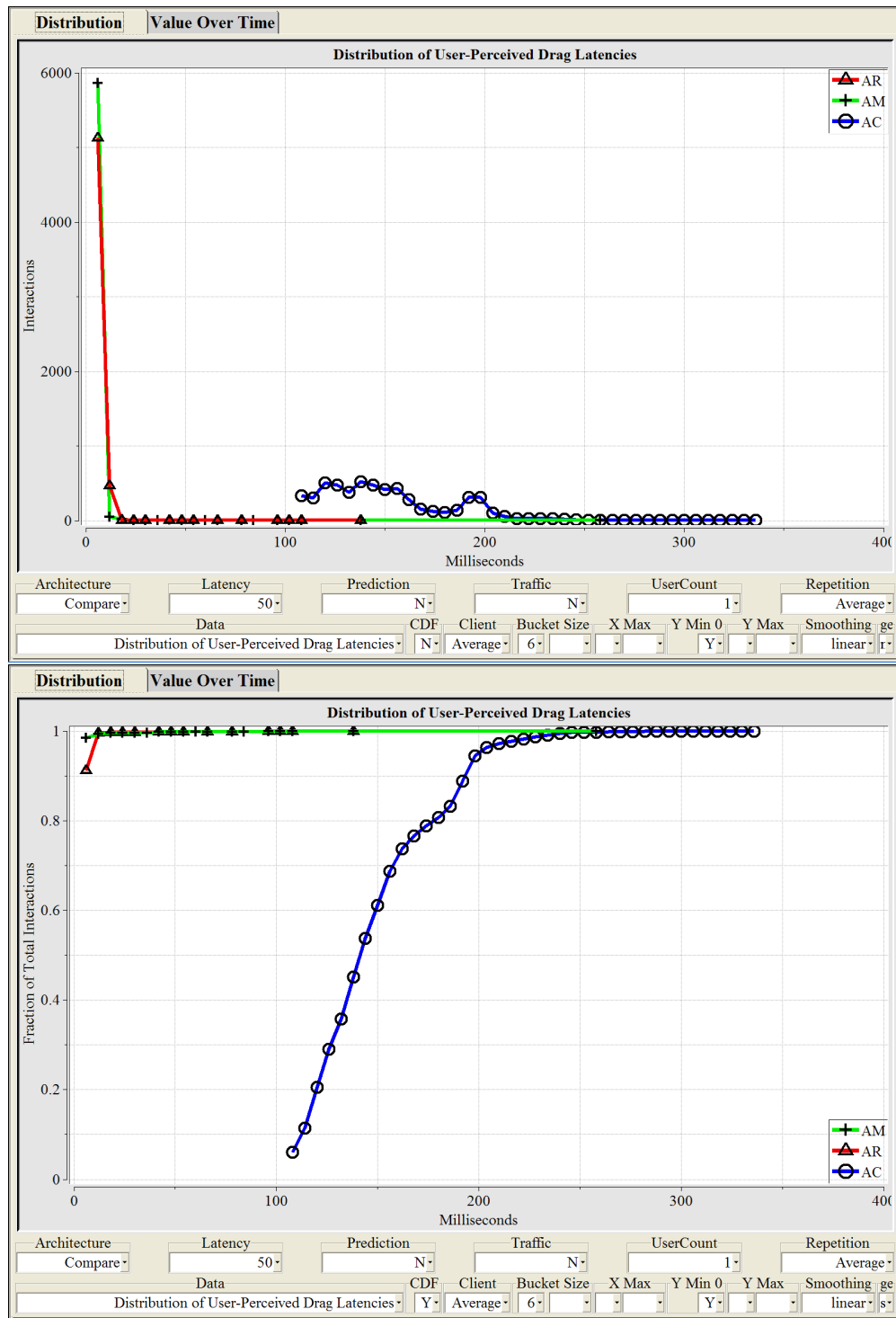


Figure 1.26: Latencies by Architecture

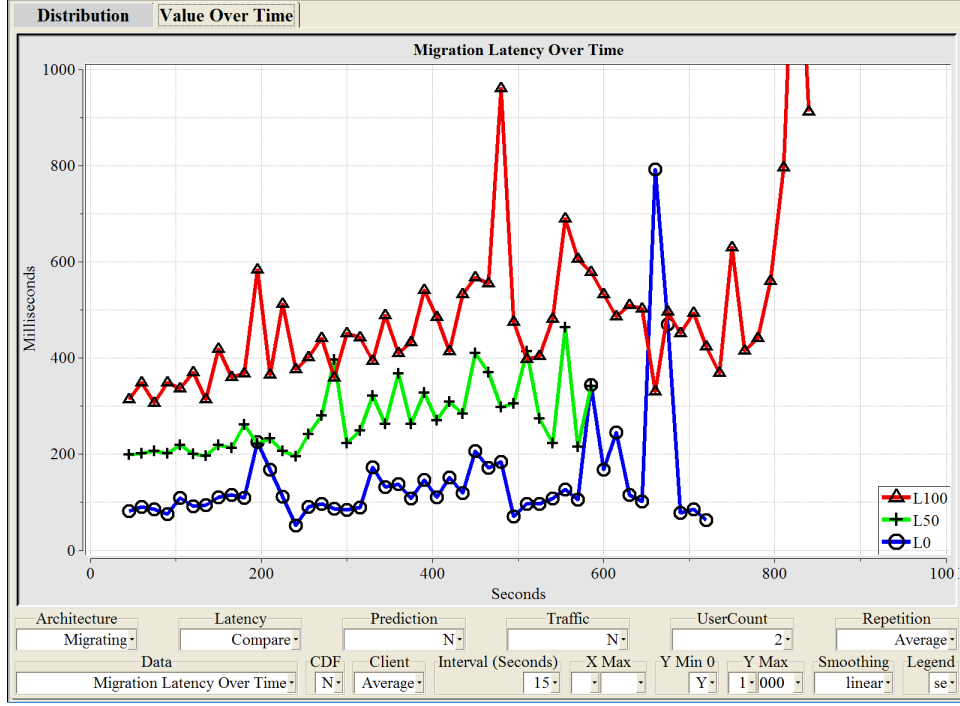


Figure 1.27: Migration Latency over Time

1.32 show that latencies typically increase and task completion times typically decrease with an increase in the number of users, for all architectures and introduced latencies. Network bandwidth utilization during interaction was better with migration than without (N messages per interaction, where N is the number of participants in a collaborative conference, instead of $N+1$ in centralized case, per Figure 1.33). This is significant when N is small, which true of most collaborative sessions. (See Figure 1.34, which only shows messages received by clients.) CPU utilization for both server (Figure 1.35) and client (Figure 1.36) were similar for all three architectures⁶. However, both server CPU and network utilization can be reduced to zero for the migrating architecture in the fairly common case where only a single user is view-

⁶The replicated architecture's server CPU utilization is artificially high, because the replication algorithm synchronized replicas through the central server.

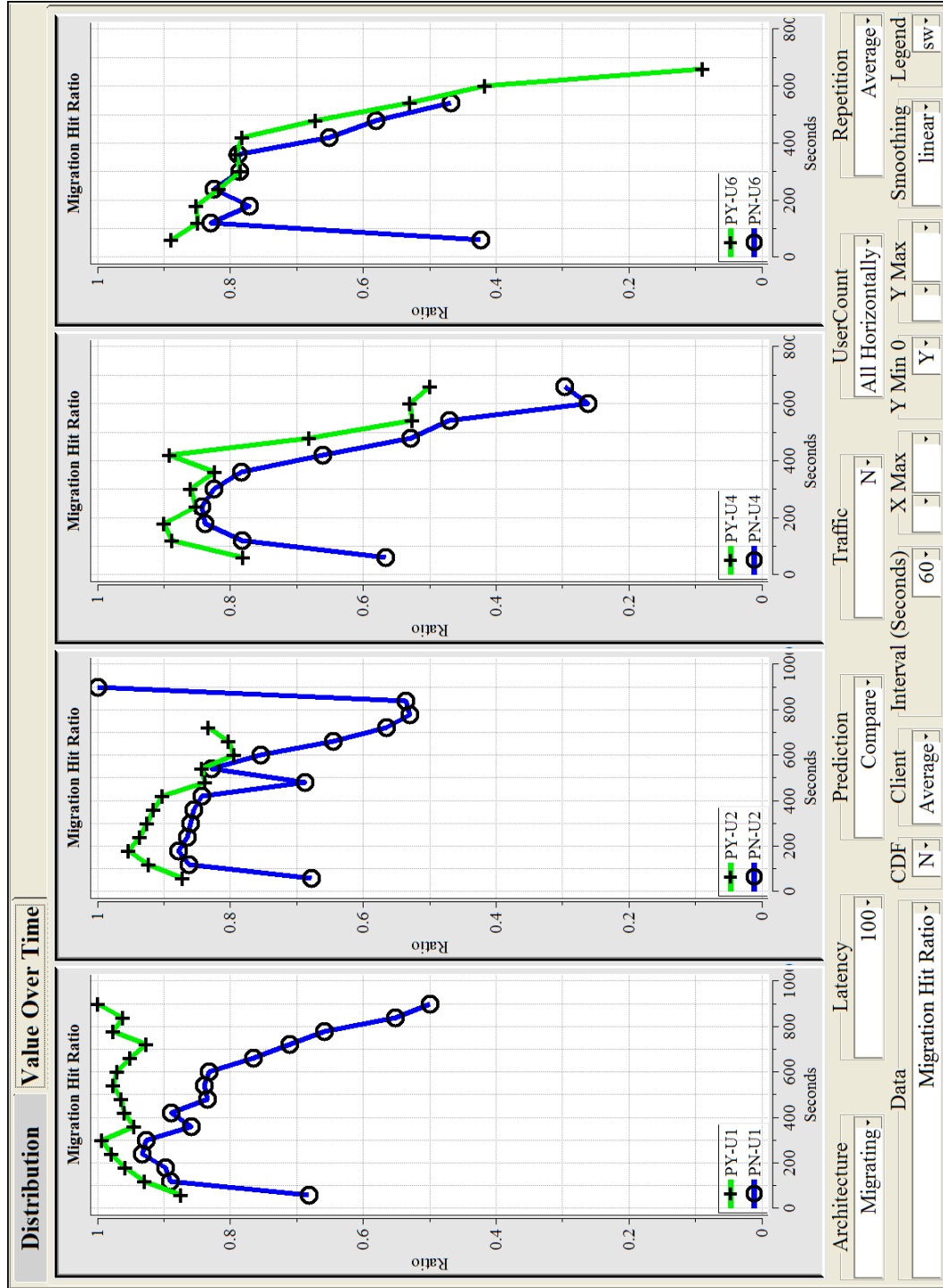


Figure 1.28: Advantage of Prediction based on Telegraphed User Intentions

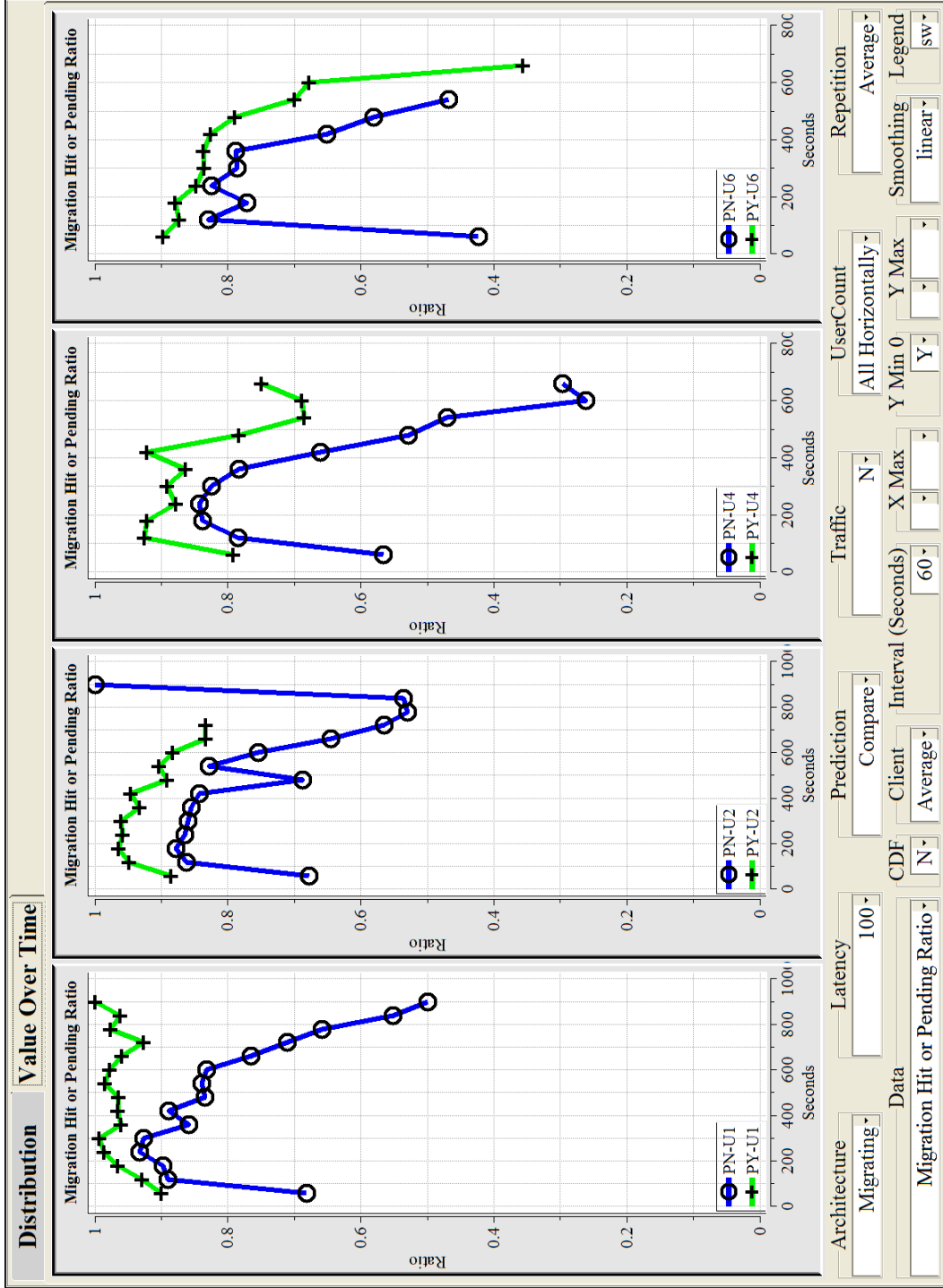


Figure 1.29: Advantage of Prediction based on Telegraphed User Intentions when Partial Migration is Included

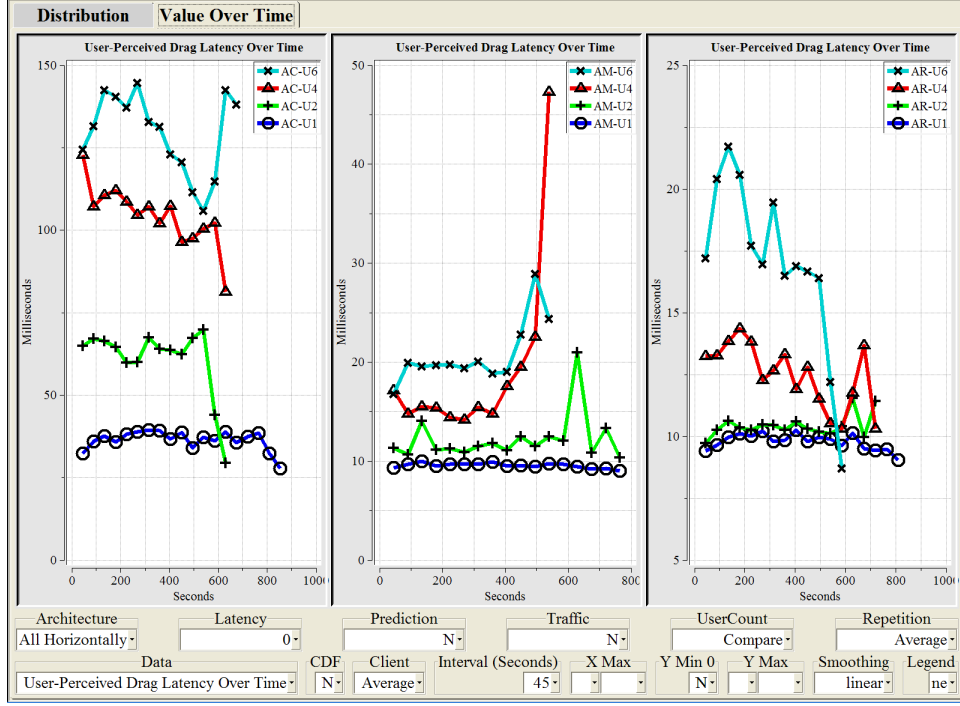


Figure 1.30: Latency and Task Duration By User Count, Latency 0ms

ing a particular entity at a particular point in time. For example, if multiple users are collaborating on a large puzzle, any puzzle pieces viewable by only one user (due, for example, to scrolling of a large workspace or document) can migrate to that user's client, and any interactions with those pieces need not use any network bandwidth or central CPU cycles. This optimization was not implemented in the prototype.

There was one important and dramatic scalability result, shown in Figures 1.37 and 1.38. These figures show that the migrating architecture scales as well as the replicated architecture, and much better than the centralized architecture, in terms on user-perceived latency as the number of users increases.

Task completion time varied a great deal due to the small size of the puzzle, the random placement of the puzzle pieces, and the independent and randomized behavior of the per-user solvers. Figure 1.39 shows the average task completion time

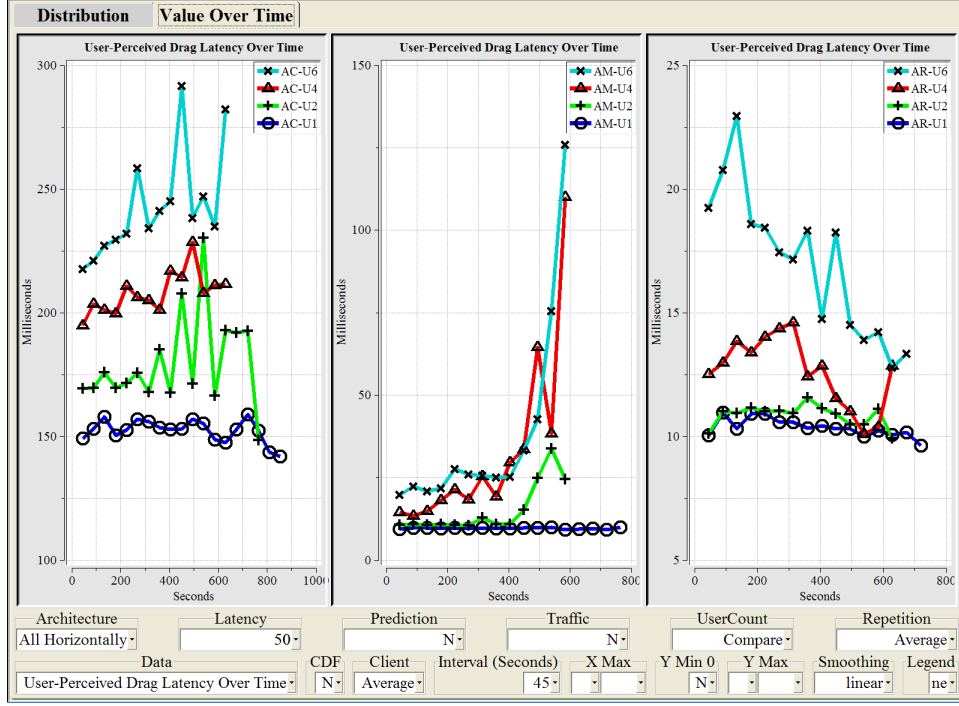


Figure 1.31: Latency and Task Duration By User Count, Latency 50ms

for each architecture, with separate averages for the migrating architecture with and without migration prediction. While average task completion time for the migrating experiments was lower than for the centralized architecture, it was higher than expected. This was due to the migrating architecture tending toward the performance of the centralized architecture toward the end of the puzzle solution when contention for pieces increases (Figure 1.40), and the short length of the experiments. In longer experiments this tendency toward centralized performance would be amortized over a longer period. The migrating architecture can also thrash on piece migrations toward the end of the solution. This effect was more pronounced in the predictive experiments due to the length of time between predictive migration and actual use of a piece (Figure 1.41). This effect could be diminished by increasing hysteresis in the migration algorithm and by migrating the piece to the physical center in order

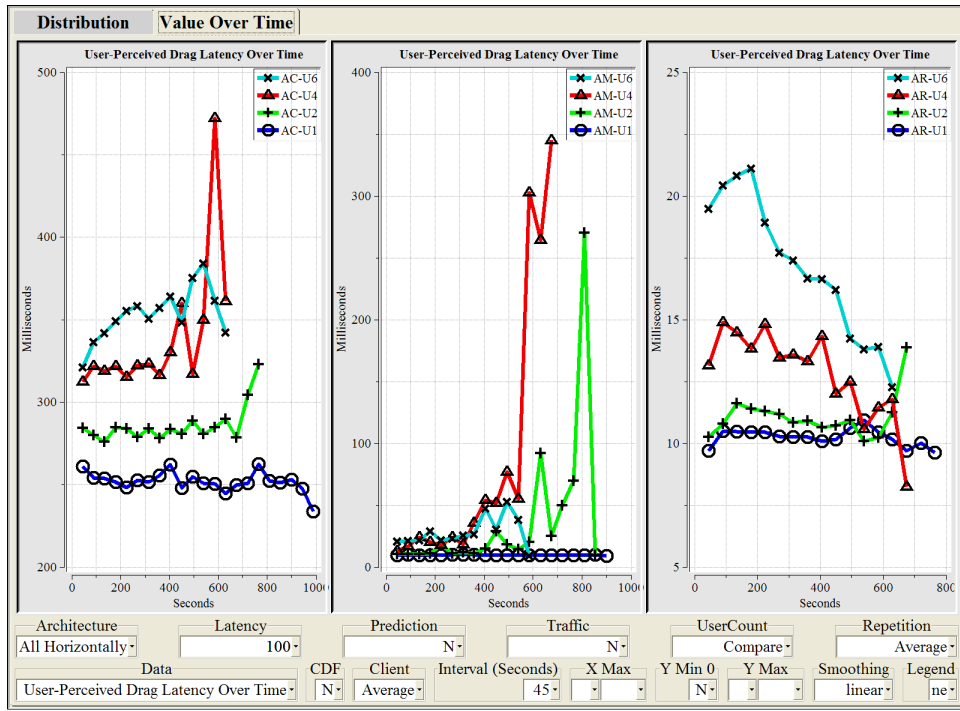


Figure 1.32: Latency and Task Duration By User Count, Latency 100ms

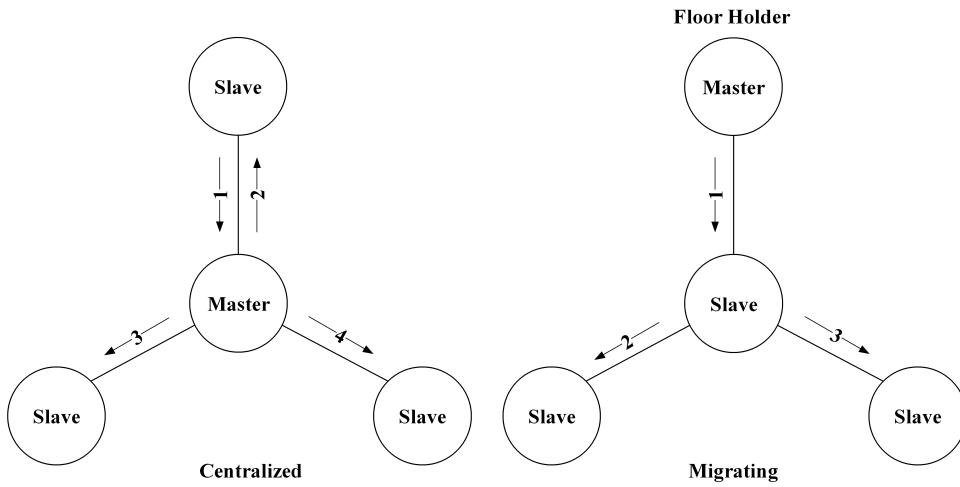


Figure 1.33: Centralized and Migrating Message Counts

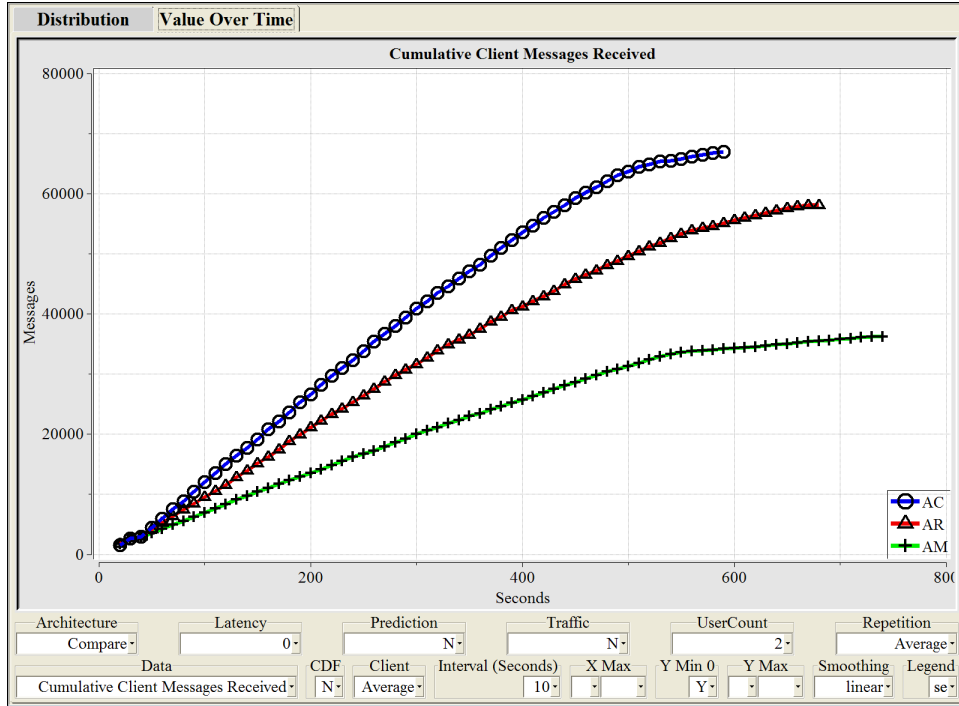


Figure 1.34: Message Counts by Architecture

to diminish thrashing. (The experiments included some degree of hysteresis, but did not migrate pieces to the center.)

Example program code is presented as evidence of the ease of construction of applications for my infrastructure. The text editor, drawing editor, and jigsaw puzzle were implemented in 322, 394, and 965 lines of code, respectively, as shown in Figure 6.2. This code was implemented for a centralized architecture, and is collaboration-unaware.

The usability of the system is argued from two standpoints. First, I demonstrated that the more complex user mental model required for replicated systems is unnecessary for my architecture, because it is fundamentally centralized. Second, I have demonstrated the utility of the entity classification with respect to identifying and implementing useful divergence scenarios.

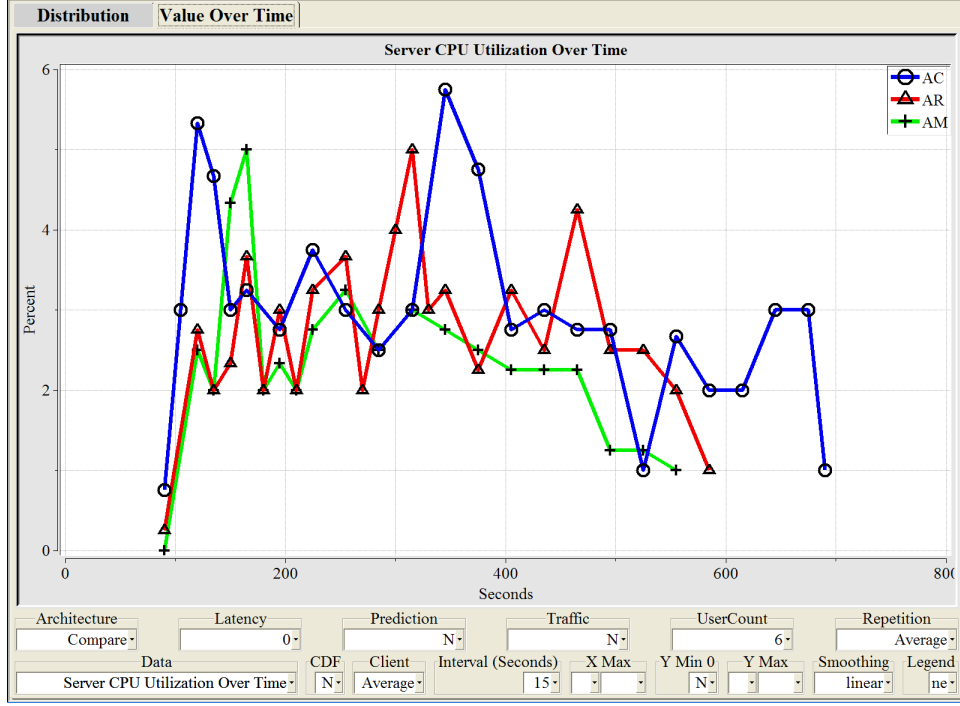


Figure 1.35: Server CPU Utilization

In sum, the most dramatic result of this work is to show that predictive, lightweight entity migration based on telegraphed user intentions reduces latency to near replicated architecture levels in what is still fundamentally a centralized architecture.

1.11 Dissertation Outline

The remainder of this dissertation proceeds as follows. In Chapter 2 I present several of the most relevant previous synchronous distributed collaborative systems, organized with respect to their fundamental approach to view computation. I then present issues with these systems regarding their functionality, implementation, and performance, based on the literature and my own observations. Finally, I present a formal analysis framework and analyze the referenced systems with respect to the

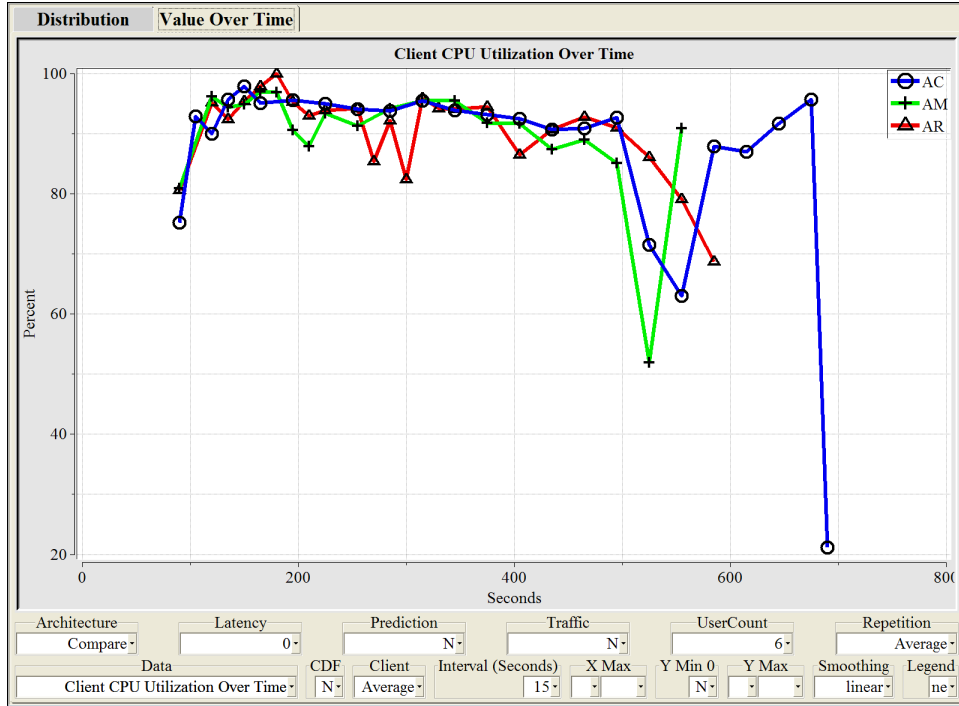


Figure 1.36: Client CPU Utilization

framework. This analysis leads to the entity classification of Chapter 3.

Chapter 3 develops an entity taxonomy based on migration characteristics, with an eye toward identifying entity classes that can be easily and quickly migrated. This classification process becomes the starting point for developing applications for my infrastructure.

In Chapter 4 I present the requirements that my infrastructure, Concur, must meet, and then the architecture of the Concur infrastructure. This chapter continues by motivating various aspects of the architecture. Chapter 5 describes a prototype implementation of this architecture.

Chapter 6 consists of my analysis and evaluation of the Concur architecture and implementation with respect to the requirements delineated in Chapter 4. I have demonstrated how Concur supports the relatively simple user mental models and

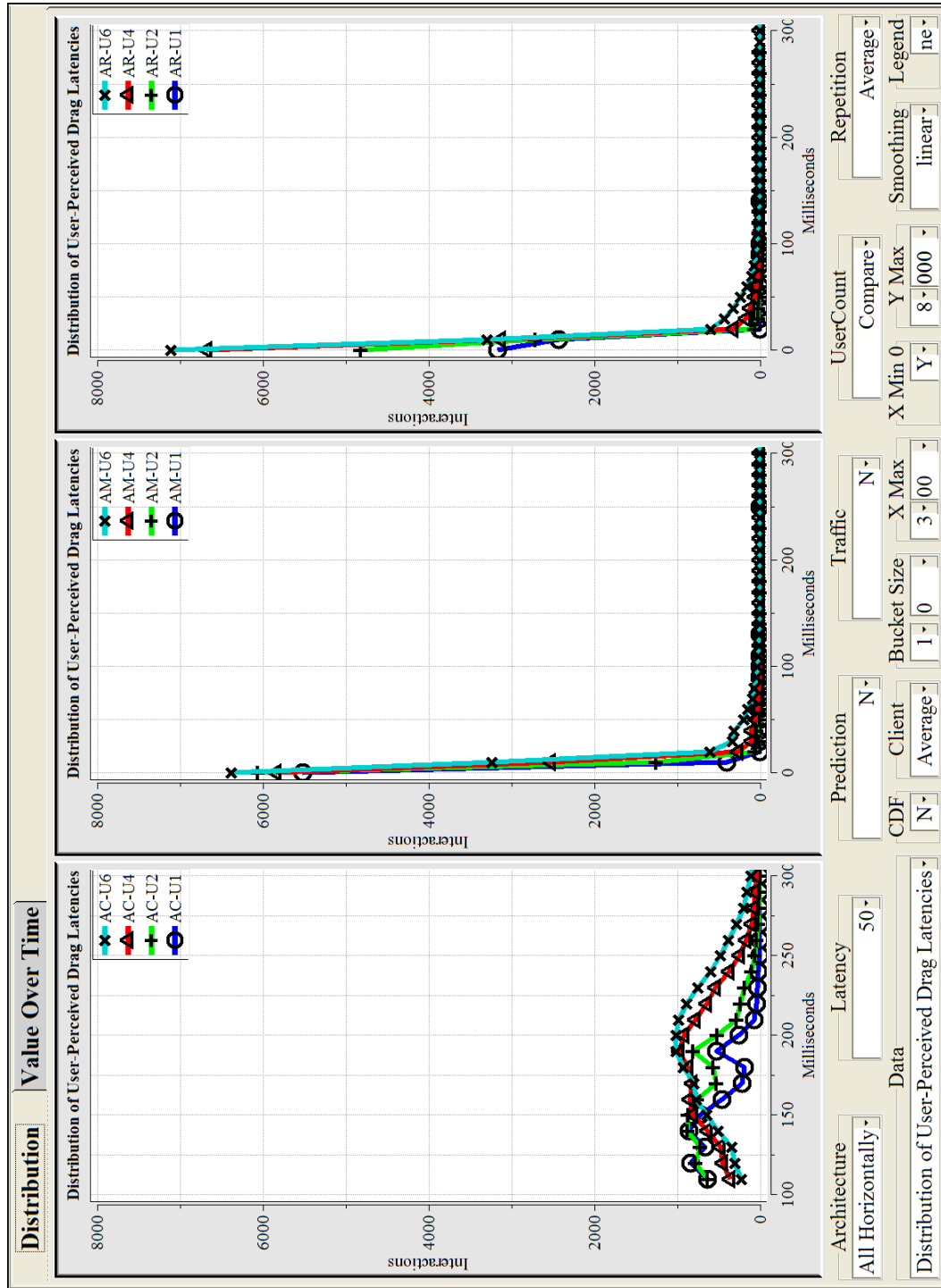


Figure 1.37: Latency Distribution By User Count

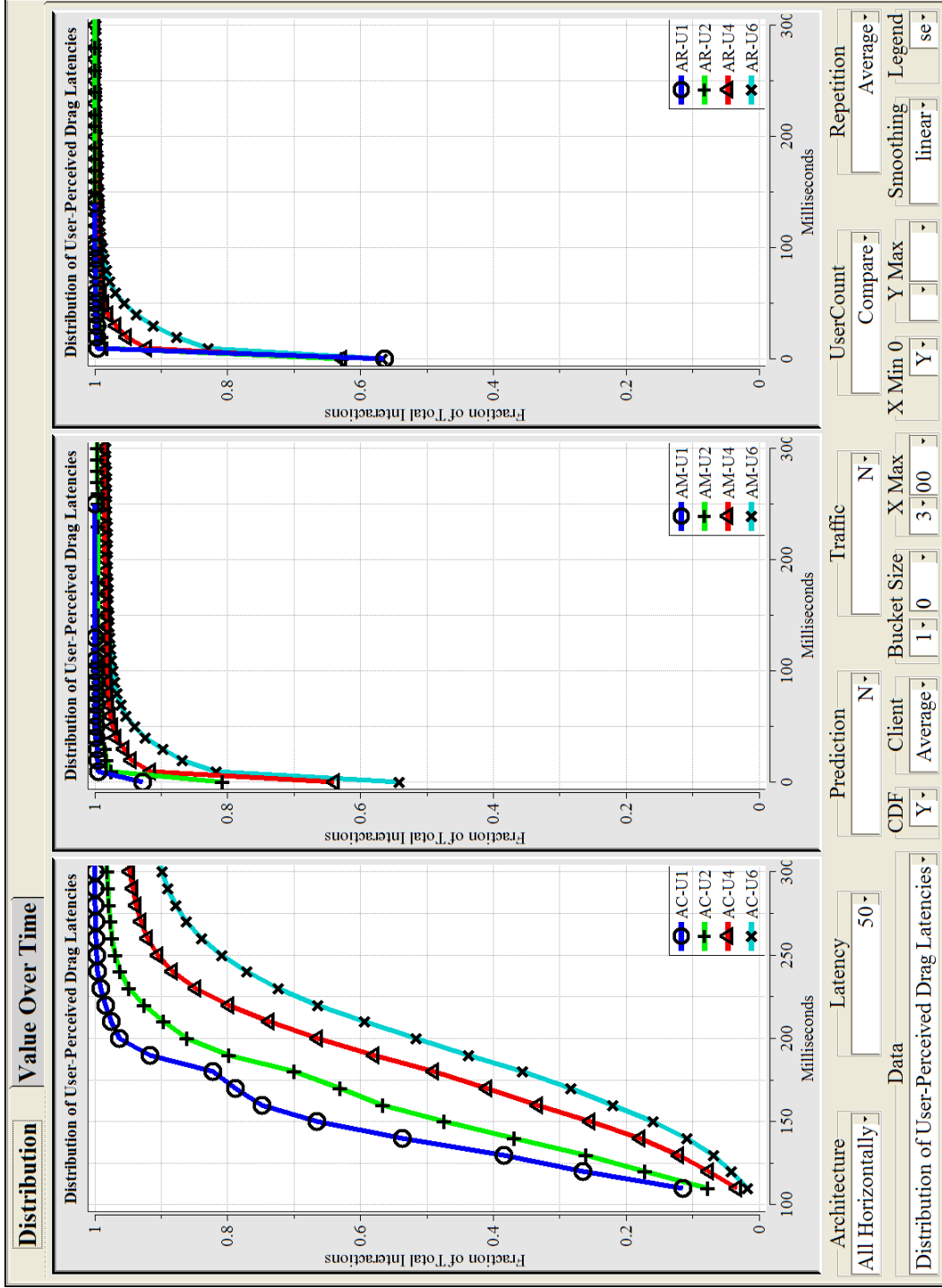


Figure 1.38: Latency Distribution By User Count

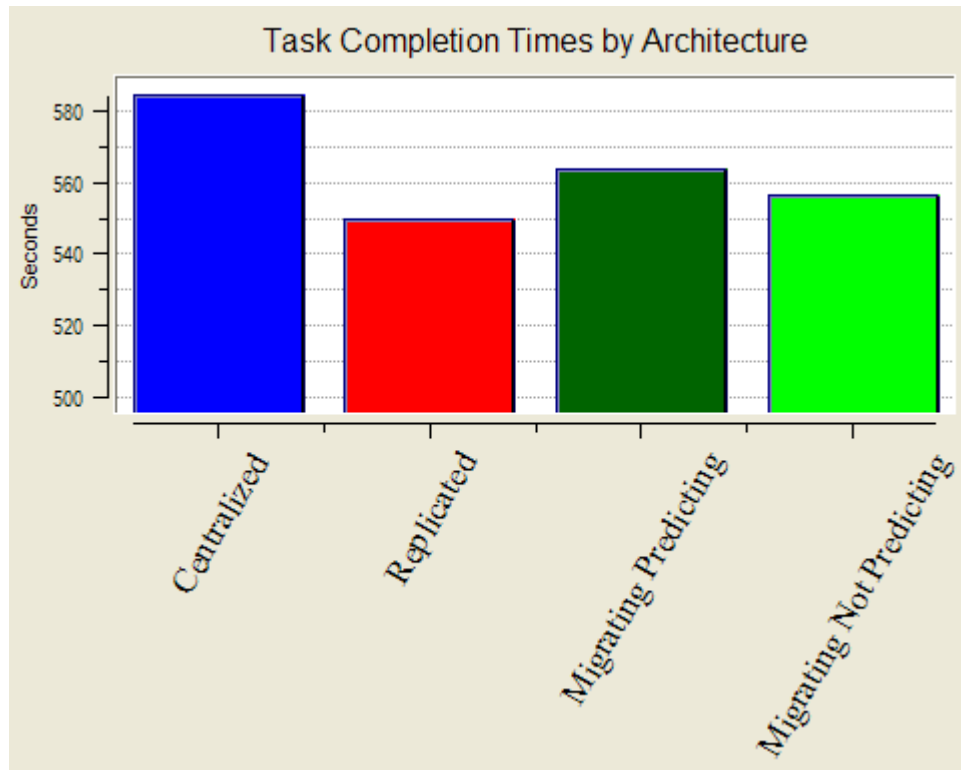


Figure 1.39: Task Completion Times By Architecture

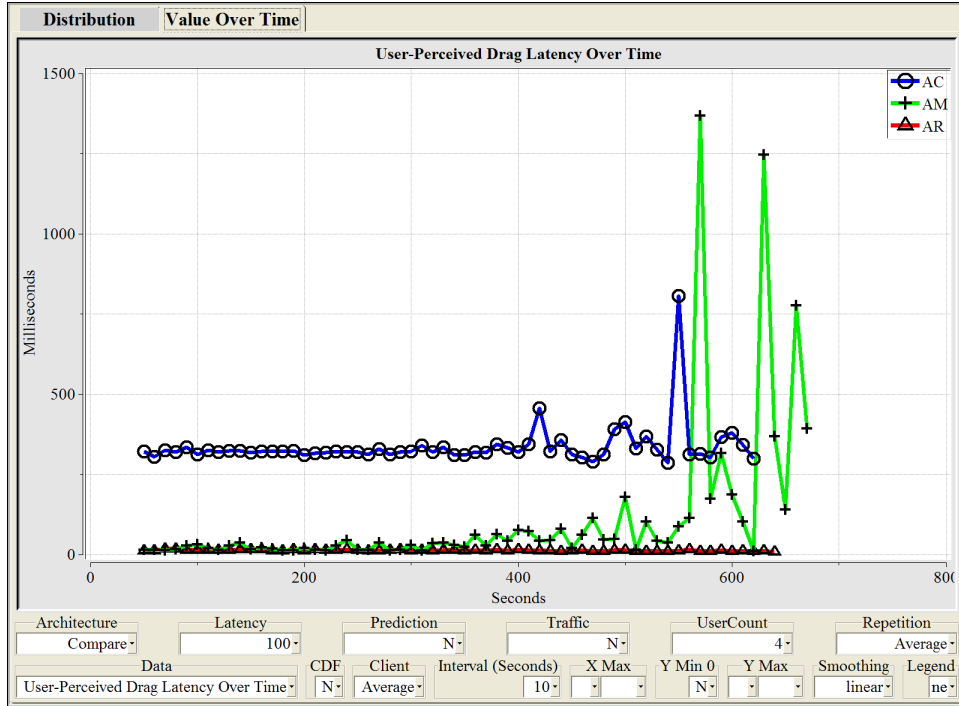


Figure 1.40: Migration Performance Under Contention

implementations of centralized systems while retaining most of the desirable latency and scalability characteristics of replicated systems.

Chapter 7 concludes this dissertation by summarizing the results of this research and pointing out interesting areas for future work.

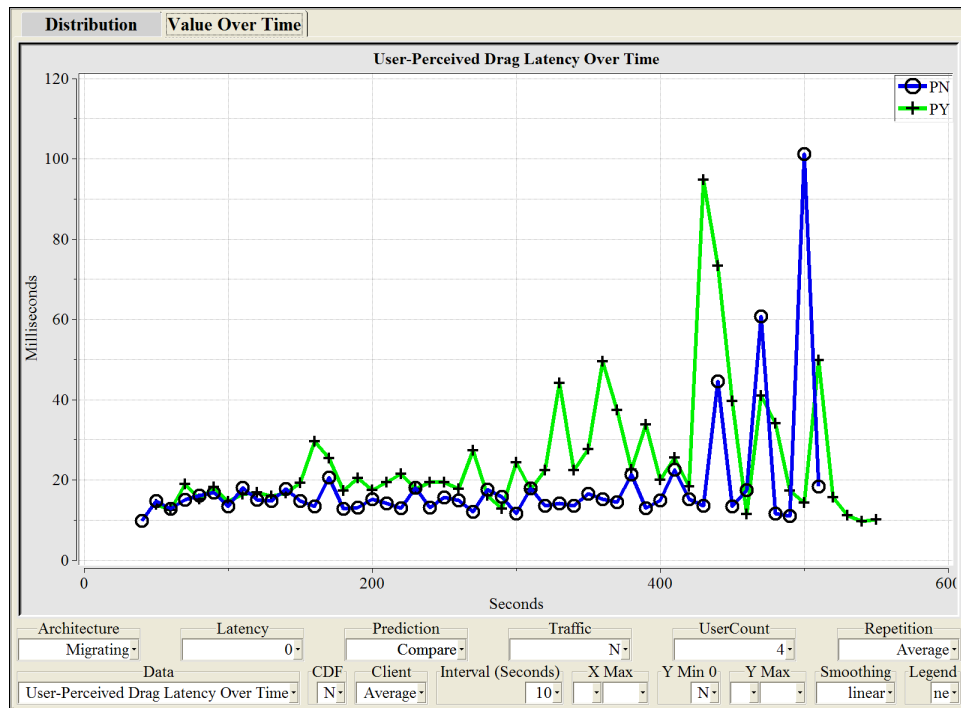


Figure 1.41: Migration Thrashing With and Without Prediction

Chapter 2

Related Work

2.1 Introduction

In this chapter I will describe several representative centralized synchronous distributed collaborative systems and will note functionality, implementation, and performance issues with these systems that are either reported in the literature or that result from my own observations. I will then provide a framework for understanding and analyzing these issues and will analyze each system within the context of this framework. Finally, I will discuss other related work that influenced the work of this dissertation in various ways. In the next chapter I will use the framework developed here to derive a novel philosophy for addressing some of the identified issues and an architecture based on this philosophy. I hypothesize that this architecture will result in collaboration infrastructures and associated applications that are easier than current systems to implement as well as applications having improved characteristics with respect to the functionality and performance issues raised.

2.2 Example Centralized Synchronous Distributed Collaborative Systems

In this section I will give a brief description of each of the following centralized synchronous distributed collaborative systems, chosen because of differences in the manner in which they compute synchronized views of model data:

- XTV and Chung's Logging Infrastructure (synchronized using state machine transitions),
- Rendezvous (synchronized using constraints), and
- Weasel and Clock (synchronized using functions).

2.2.1 XTV and Chung's Logging Infrastructure

XTV[AWF91] (Figure 2.1), a window sharing system for the X Window System[SGR92], was developed at the University of North Carolina and Old Dominion University.

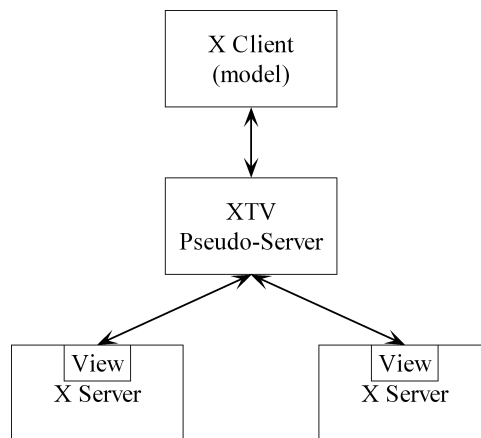


Figure 2.1: XTV

XTV is implemented as a pseudo-server that intercepts the communication protocol

between an X client (application) and server (window system). Thus, it is a client of the server(s) and a server to the client. At a very high level of abstraction, XTV “simply” distributes messages from the client to multiple servers, and merges messages from the servers to the client. In this way it can enable multiple people to interact with a single-user application. These applications are designed to receive input from only one user at a time, so XTV must also implement *floor control* to coordinate input from multiple users.

The X window system does not directly support model/view separation. In Figure 2.1, the X client, which is a black box to XTV, represents both the model (application state and behavior) and the view (a mapping from the model to a particular projection represented as X windows). In the spirit of the Observer design pattern, where subjects and observers can be chained, we will consider what is logically the view (the user interface computed by the X client) to be the model to be shared by XTV. We will then consider the view to be the portion of the X server that maps this user interface to the projection realized as windows on a display. This view can be conceptualized as a state machine where most state transitions are triggered by messages from the application via the pseudo-server, though some can also be triggered by the X server itself or by other clients (such as window managers). The model, implemented by the X client, is also assumed to be a state machine responding to X input events and request responses, and possibly other stimuli. The domain of this model is unusual because the semantics are those of a user interface, which is typically the responsibility of views.

Over time, XTV was extended to support *latecomers*[CJAW93][CJAW94] (users joining a conference after it has begun) by logging the requests made by the X client and replaying them to the X server of the new participant (Figure 2.2), in effect, replicating the view. After the log has been replayed, the new participant is treated

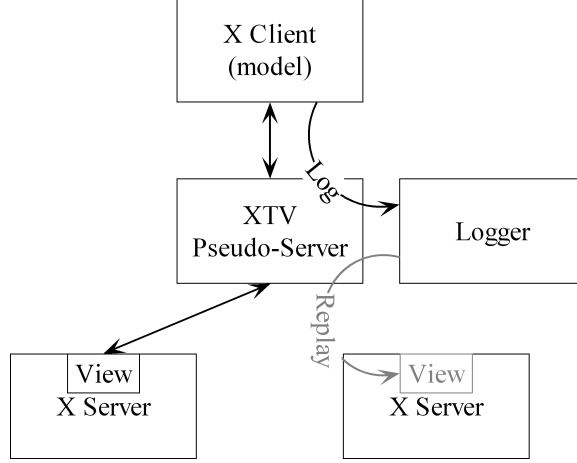


Figure 2.2: XTV Latecomer Support

the same as a pre-existing participant. While the log size is theoretically $O(r)$ where r is the number of requests (a measure of the length of the conference, which grows indefinitely), XTV compresses the log such that, except in unusual circumstances, it approximates $O(s)$ where s is the size/complexity of user interface, which tends to be relatively stable.

Later, similar mechanisms were used to migrate X applications from one host to another[CD96], by logging application inputs and replaying them to a new application instance (Figure 2.3), in effect, replicating the application. The original instance can then be terminated (for migration), allowed to continue running (for replication), or kept in a dormant state so that it can be re-activated later after its state is updated through a log replay. This application migration process is less deterministic than replicating the view. Applications are, in general, less deterministic than window servers with respect to their state transitions in response to replayed logs, because applications are more likely to respond to other stimuli such as the passage of time and other external references (e.g., files). In addition, far less is known about the semantics of various applications than window servers. Nevertheless, the XTV re-

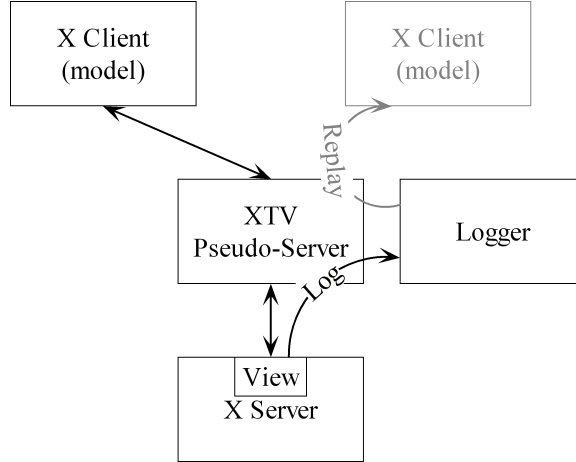


Figure 2.3: XTV Client Migration

searchers were able to get migration to work in most cases. However, logs are not as easy to compress in the input direction as they are in the output direction. This is because one never knows what an input might do to an application state, while window system operations are well defined. Thus, nearly all inputs must be replayed, and the size of the log is $O(i)$, where i is the number of inputs.

These log-based view and application migration facilities were later generalized to apply to any collaborative system involving layers represented by state machines, where the state machine inputs can be captured and replayed[CDR98]. Each system must be adapted to the log-based approach by providing a *loggable*, which is a component that intercepts inter-layer, system-specific protocols, translates them to and from the generic protocol used to communicate with Chung’s logger, and replays them to the adjacent layer. A range of log compression capabilities can be achieved by providing more or less semantic information describing the messages passed to the logger. View and application migration operations were then used to create an infrastructure capable of dynamically changing the architecture of any such collaborative system among centralized, replicated, and hybrid variants[CD01][Chu02][CD04].

XTV does not support different views of the same model. Chung’s infrastructure does not directly support such divergence either, though if it replicates components at a high level of abstraction, lower levels are free to diverge in this way.

2.2.2 Rendezvous

Rendezvous[HBR⁺94] (Figure 2.4) is a language and centralized architecture for synchronous distributed collaborative applications, developed at Bellcore. It is de-

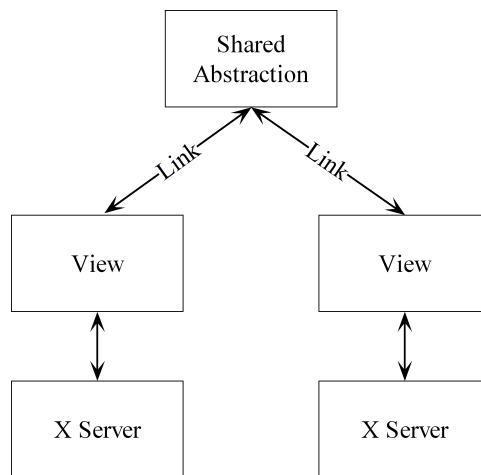


Figure 2.4: Rendezvous

signed according to the Abstraction-Link-View (ALV) paradigm[Hil92][BH93], an adaptation of push MVC where the abstraction corresponds to the model, the view subsumes the view and controller functions, and abstractions and views are synchronized via links.

Abstractions, links, and views are represented in the Rendezvous object-oriented language, an extension of the Common Lisp Object System (CLOS)[CLO08] supporting events and constraints. Links are objects encapsulating bundles of one-way constraints between abstractions and views. Constraints are typically paired, with one in each direction. They synchronize abstractions and views by linking individual

state variables in abstractions with corresponding variables in views, and vice versa. Links may perform arbitrary transformations of the data along the way, enabling abstractions and views to each represent the same logical data in a manner that is convenient for the implementation of their respective responsibilities.

Abstractions and views are tree data structures, typically with a high degree of correspondence between the structure of a view and the structure of a corresponding abstraction. This simplifies the correspondence between abstractions and views maintained by links. In turn, the view tree corresponds to the containment hierarchy of user interface components. Thus, by transitivity, there is generally a strong correspondence between the visible user interface and both the view and abstraction structures. A special kind of link called a tree-maintenance link is used to maintain the appropriate constraints between abstractions and views as the structure of the abstractions and views changes.

The view tree declaratively specifies the user interface, which is actually maintained by the Rendezvous infrastructure using the X Window System. The Rendezvous infrastructure maintains constraints on the declarative view data structure, responding to changes in this structure by updating the visible user interface. In the other direction, interactions with the user interface trigger events which are directed to event handlers in view code. These may cause changes to view state variables, which may be the source end of constraints targeted at variables in the shared abstraction. Thus, the abstraction is modified in response to user events, triggering abstraction-to-view constraints in the same or other views. In this manner, views are kept in sync with the abstraction and therefore with each other.

Different views of the same abstraction can, of course, be implemented by different links and/or different view code. In addition, not all view state needs to be linked to abstraction state, which allows individual views to diverge from other views, even

given identical view code and constraints.

Rendezvous is strongly centralized, in that models and views both reside on a centralized host. Views are then distributed using the X Window System. This architecture greatly simplifies the constraint maintenance engine, which does not have to be distributed. The Rendezvous researchers also developed an experimental version of Rendezvous that distributed view computation using a distributed subset of their constraint facilities, but a fully distributed version of Rendezvous was never completed.

2.2.3 Weasel and Clock

Weasel[Urn92][GU92][UN94] (Figure 2.5) is a development environment supporting synchronous distributed collaborative applications, from GMD and the Norwegian Institute of Technology. It uses an adaptation of the pull MVC paradigm.

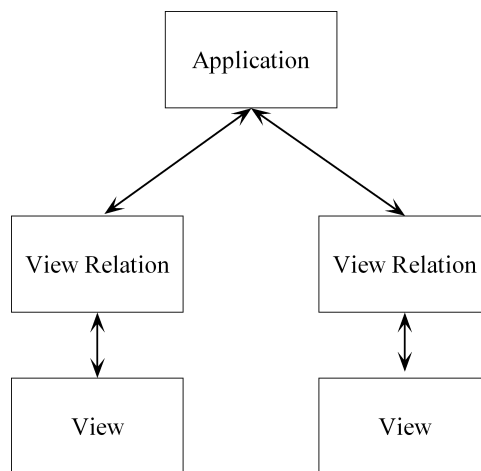


Figure 2.5: Weasel

In Weasel, applications written in an imperative language (Turing) play the role of models. Views are constructed using the Relational View Language (RVL), which creates a bi-directional relationship between application data structures and views.

When constructing views, functions (in the functional programming language sense) map from data structures in the application to view (user interface) components. Functions can be composed to produce composite views. Functions can also be annotated such that the view components they produce (e.g., buttons) are active. If such an active component is manipulated (e.g., a button is pressed), the Weasel infrastructure stores a value in an argument of the function. This value can then be propagated back to the application. Thus, as in Rendezvous, views subsume the controller MVC function. Changes to application state trigger re-evaluation of the view function. In the course of re-evaluating this function, expressions are passed to the application for evaluation, so that the new view will correspond to the current application state.

Parameters to Weasel view functions can be either remote (i.e., they can come from the application) or local (from the view component itself). Thus Weasel, like Rendezvous, can implement different views of the same application by using different functions or local view state.

The authors of Weasel later developed the Clock architectural style and visual architecture language[GU96], and the associated ClockWorks[GMU96] visual programming environment. The components of a Clock architecture are programmed textually using a functional language based on Haskell[Bir98].

Clock architectures are designed visually using ClockWorks. On the left side of Figure 2.6 is a simplified drawing of a Clock architecture being designed in ClockWorks. On the right side of the figure is a drawing of the resulting user interface. This simple application represents a toggle switch labeled **Press** below a light that goes on or off each time the button is pressed (it is off at present). A Clock architecture is a hierarchical structure matching the user interface containment hierarchy. Components are represented by rectangles, with a component name at the top of each rectangle and a class name at the bottom.

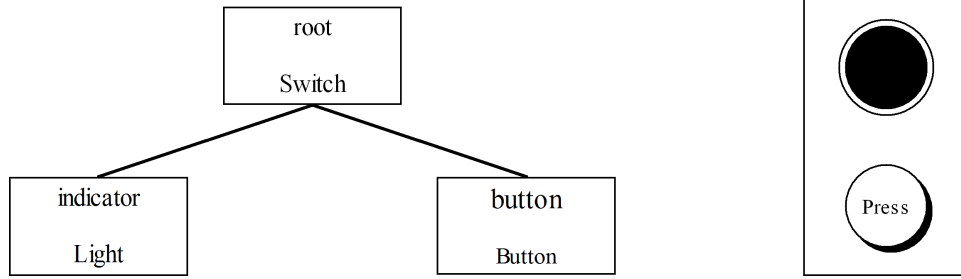


Figure 2.6: Clock Architecture Example

Figure 2.7 shows more detail of this simple clock architecture¹. Boxes above components (Toggle and Depressed, in the figure) are Abstract Data Types (ADTs) associated with the components. Arrows to the left of components and ADTs denote the supported incoming event/update (single arrow) or request (double arrow) calls. Arrows to the right are outgoing calls that can be made by the respective components. Identifying these outgoing calls supports the Clock constraint maintenance system, described below. Rounded boxes show the code for the indicator (left) and button (right).

Assume the user depresses the button. When he does so, the *mouseButtonUpdt* function is evaluated with the “Down” argument. It calls *depress*, and this call bubbles up the tree to the Depressed ADT, changing its state. Since the button view function calls *isDepressed*, a constraint invokes the *view* function of the button, and the button is redrawn with a sunken relief. When the user releases the button, *mouseButtonUpdt* is called with the “Up” argument. This calls *release*, which bubbles up to the Depressed ADT again and resets its state, triggering the view function of the button to be re-evaluated, restoring the button to raised relief. Then *mouseButtonUpdt* calls *buttonClick*. This bubbles all the way up to the Toggle ADT, toggling its value. Since the indicator view function calls *getValue*, a constraint causes the re-evaluation

¹I have taken some minor liberties with the language in order to clarify the example.

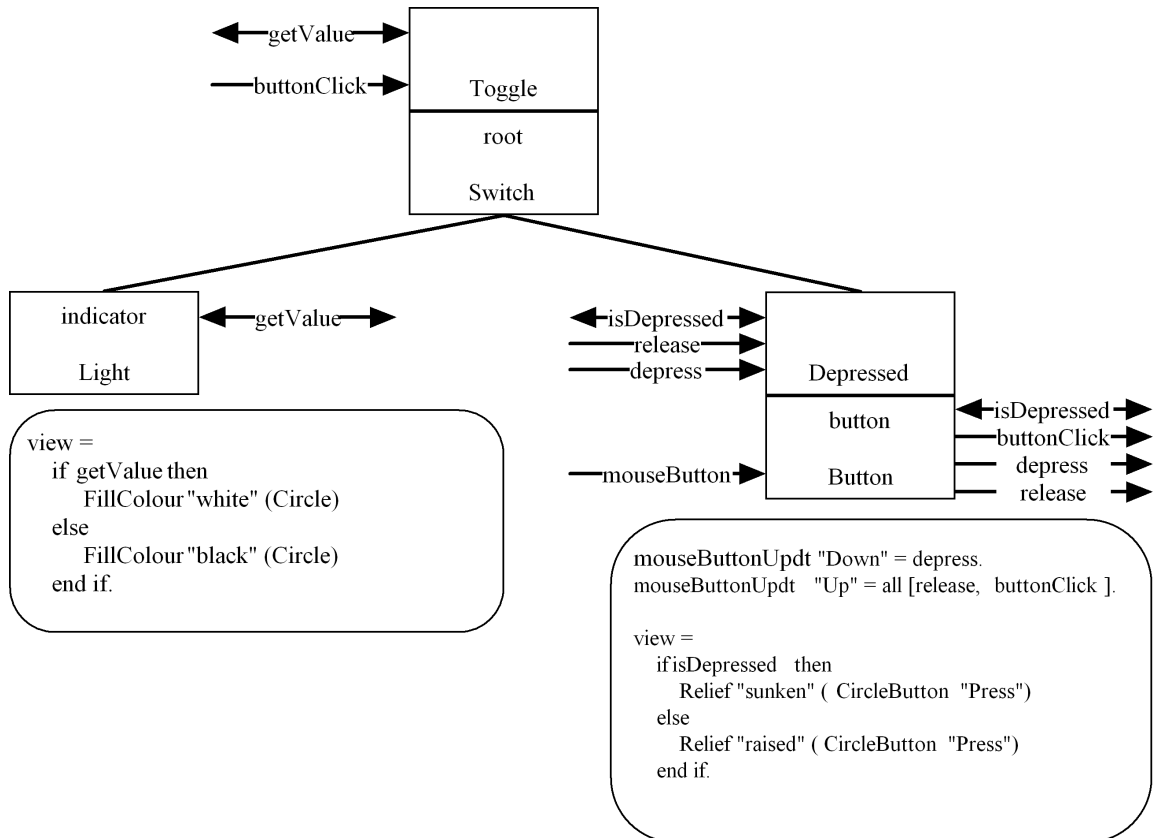


Figure 2.7: Detail of Clock Example

of this function, which turns the light on by filling the upper circle with the color “white”.

Clock’s call bubbling facility enforces data hiding because components can only make calls on components and ADTs that are their ancestors in the tree. ADTs should therefore be located at the lowest point in the tree covering all components accessing the ADT. Clock’s constraint facility enables communication among components that are not in an ancestor/descendant relationship. Both facilities are indirect means of communication, where components do not directly reference each other. This facilitates reorganization of the architecture, which can be performed in ClockWorks by direct manipulation. Unlike XTV, Rendezvous, and Weasel, models are distributed throughout Clock architectures as ADTs, making the various Model/View relationships difficult to diagram.

Clock supports various annotations to the architecture, telling ClockWorks how to generate a particular implementation. Like Rendezvous, default implementations of Clock architectures are purely centralized, with views distributed using the X Window System. One annotation tells ClockWorks to divide the architecture into a client/server implementation (Figure 2.8). Various caching algorithms can be selected with this annotation. In the figure, the portion of the architecture above the annotation would be centralized, while the portion below (a copy of which would be created for each participant) would be distributed. Calls from client to server and constraint invocations from server to client will cross the network transparently. Clock annotations are (normally) semantics-preserving, meaning that only the implementation, not the functionality of the system is changed by an annotation. This allows applications to be developed without annotations and then optimized using annotations. Other annotations specify the type of concurrency control to use on particular ADTs, and which ADTs should be replicated. Replicated ADTs may be specified on the server

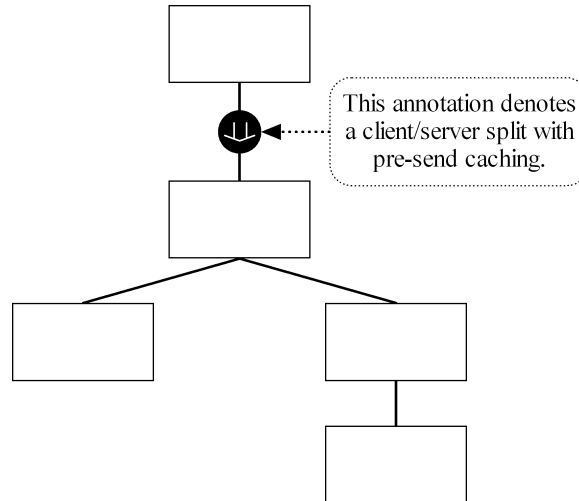


Figure 2.8: ClockWorks Client/Server Annotation

side of the diagram to denote more global call/constraint bindings, but replicas will be created for each client. A centralized replica will also be created to distribute updates from one replica to another and to support latecomers. (When ADTs are replicated, the developer asserts that the concurrency control implemented will not change the semantics of the application, since this is difficult to determine automatically.)

Like Weasel, Clock uses a pull MVC variant. However, some of the caching options implement push-like semantics, where the server anticipates model values that may be needed in the future based on past queries, and proactively pushes these values to the client cache[GUN96a]. Like XTV and Chung’s infrastructure, but unlike Rendezvous and Weasel, Clock provides no support for different views of the same model.

2.3 Issues with the Above Systems

This section will briefly discuss issues with the above systems at a very high level. These issues will be discussed in more detail in the remaining sections.

2.3.1 Functionality

There are several major functionality issues with the foregoing systems that I will enumerate here:

MVC Issues

MVC issues are discussed here in the functionality section rather than in the implementation section because the difficulties encountered are architectural, and they bleed through strongly to the functionality of the system, as described below.

Rendezvous and Weasel both combine the view and controller functions of the MVC paradigm. XTV and Chung's infrastructure only marginally support MVC through their model/view separation in layers. This is because models are black boxes and cannot be queried, which is a required operation for pull MVC. Also, XTV and Chung's infrastructure rely on counting messages from models in order to heuristically determine whether models are synchronized. This does not work with push MVC, where different notifications are sent to different views, depending on view interest. (See [Chu02] for a more complete description of the difficulties Chung's infrastructure has in supporting MVC.) Only Clock has a separate controller mechanism, in the form of event handler functions that map between operations at the user-interface and application domain levels (see [PG99] and Figure 2.7). However, Clock uses the less efficient pull paradigm, enhanced with caching facilities as described above to mitigate some of the performance problems. A push approach would be more efficient, but it is difficult to implement with a constraint system based on method calls (such as Clock) rather than events.

Rendezvous advertises that the ALV paradigm provides better separation of models and views than MVC, because links can map between the two, eliminating the

view’s knowledge of the model. Clock’s indirect call and constraint mechanisms advertise similar separation via indirect communication. However, both systems tend to force model data to be structured like the user interface. This complicates implementing multiple, differing views of the same model, since the model cannot simultaneously reflect the structures of radically different views. In general, any model/view dependencies tend to reduce the reusability of these components, which diminishes divergence possibilities available to participants.

Weasel implements a fairly traditional pull MVC paradigm, where view and controller are combined. The Rendezvous ALV paradigm is more like push MVC. Rendezvous and Weasel both modify models directly at a low level of abstraction, rather than at the level of application-domain operations. Operating on models at a low level of abstraction tends to complicate concurrency control on models (which may need several low-level operations to be performed atomically), and consequently forces user-interface and collaboration awareness into the model. This makes it difficult to reuse models in unanticipated ways. Low-level operations on models also imply a more detailed knowledge of models by views (or links, in the case of Rendezvous).

Unshareability of Local State

Rendezvous, Weasel, and Chung’s infrastructure² allow view-local state. Rendezvous and Weasel provide no mechanisms for sharing this state. This means that any application using this mechanism to diverge cannot share all of its interaction with anyone else. For Chung’s infrastructure, whether local state can be shared or not depends on the layer at which logging takes place. If logging takes place at a higher abstraction layer than the local state, then that state cannot be shared unless

²... and maybe Clock, depending on whether ADTs can be represented below the client/server divide, and if so, what the semantics of that would be.

low-level events describing changes to that state are passed to higher layers. For example, mouse cursors are not shareable in XTV unless the window system is told to send all cursor motion events, because otherwise the local state of the cursor position is entirely within the X server.

Invisibility to Participants of Divergence Parameters

Neither Rendezvous nor Weasel provides mechanisms for exposing to the users the various ways in which their views might be allowed to diverge. For Rendezvous, this would be difficult, because constraints link low-level variables that may not have meaning to the user. For Weasel, it might be more reasonable, since function parameters can operate at a higher level of abstraction. However it is not clear that function parameters would always indicate a reasonable point of divergence, or how to indicate which parameters do and which don't. Thus, for these systems, specifying divergence is mostly within the realm of the developer, not the participant. XTV, Chung's infrastructure, and Clock do not support much in the way of divergence.

Non-determinism

XTV and Chung's logging infrastructure are, in general, non-deterministic. They are implemented using heuristics that usually work, but are not guaranteed to do so. This is not a fault the infrastructure, as it is doing all it can to implement synchronous distributed collaboration using components not necessarily designed for this purpose. Nevertheless, non-determinism is a serious deficiency, and any collaborative infrastructure should be deterministic whenever possible.

2.3.2 Implementation

In this section I will point out two issues with the above systems that complicate infrastructure and/or application implementation: state machine problems and container requirements.

State Machines

In general, moving or copying state machines and keeping state machine replicas up to date can be both inefficient and difficult to implement, as is attested to by the literature on XTV and other window sharing systems and Chung’s logging infrastructure[LJLR90][Pat90][CJAW93][AWJ94][CD96][CDR98][Chu02]. This applies to both model state, which is usually unavoidably represented by a state machine, and view state, which often is but need not be represented by a state machine. It also applies to local state in view components that update other state using constraints or functions. Difficulties include:

- **Divergent containers.** Usually replica containers (execution environments) are mostly, but not entirely, identical. Examples are differences in available fonts and color representations in the X Window System, applications or infrastructures installed on some systems but not others, and applications or infrastructures installed to different revision levels.
- **Uncontrolled inputs.** Generalizing on the divergent container issue, it is very often the case that state machines have inputs which are not under the control of the infrastructure attempting to set the component’s state. Examples are files, environment variables, and interrupts. This can cause non-deterministic state changes, making it difficult or impossible to ensure that the desired state has been achieved. Non-deterministic state changes are often impossible to detect,

and their effects on a component's behavior can be hidden until arbitrarily distant times in the future.

- **Timing dependencies.** A special case of uncontrolled inputs is timing dependencies (where time is the uncontrolled input). Timing dependencies can also lead to non-determinism. They are much more difficult to control than other inputs.
- **Non-idempotent operations.** State machines not only change state in response to inputs, they also emit outputs. Often, especially in the case of models, the outputs they emit are non-idempotent. Writing to a file, sending email messages, and communicating with external processes are three examples. A more subtle example is making changes to a display that are visible to a user, since the user is capable not only of observing the current state, but of remembering the sequence of observed events. Non-idempotent operations in general can occur when a state machine changes state that is outside the bounds of the state machine itself, which it therefore cannot fully control.
- **Representation mappings.** Logically identical state in two replicas may require physically different representations. This requires representation mappings to be performed when creating or updating a replica. Examples are resource identifiers and color specifications in the X Window System.
- **State accessibility.** It may be difficult or impossible to retrieve or set the desired state.

Of the systems described, XTV and Chung's infrastructure manipulate state machine components. Rendezvous and Weasel³ also have local state which must be

³and maybe Clock

treated as a state machine.

Container Requirements

Container requirements are critical in distributed systems, whether the components executing in these containers are represented as state machines or not. In distributed systems, container requirements go beyond the obvious need for containers to provide all the facilities and resources required by a contained component. Distributed systems imply multiple, independent execution environments (containers). Infrastructures must have guarantees about these containers in order to ensure that the operations they perform will have the specified effect. At a minimum, the infrastructure must be able to detect container properties that may differ from system to system. Ideally, containers should make guarantees of identity. Two useful methods that help achieve such guarantees are to ensure that the container requirements are extremely simple and well-defined, and to restrict inputs to a container to a minimum. Security and idempotency concerns motivate restricting container outputs to a simple, well-defined minimum as well.

Another strategy for achieving container identity is to distribute sub-containers from a central location. For example, if the operating system satisfies the necessary guarantees as an outer container, portable application code can be delivered from a central location to ensure that the application-level container is identical at all sites, rather than depending upon the operating system to have the proper application at the proper revision level installed.

The container requirements of the systems described above are as follows:

- **XTV:** For each model container, it requires an operating system with the same version of the X client application installed, any external resources the client

might need, and TCP/IP. For the view container, it requires an X server. Any differences in X servers are discernable using the X protocol. The infrastructure (pseudo-server) only needs an operating system supporting TCP/IP and the infrastructure code itself.

- **Chung's infrastructure:** For each model container, it requires an operating system with the same version of the application installed, any external resources the application might need, and TCP/IP. The view container requirements depend on the system being logged. The infrastructure requirements are the same as XTV's.
- **Rendezvous:** The central container requires the Rendezvous infrastructure, model and view code, any external resources the model code might need, and access to an X server. Since the model and views are not distributed, this is only required on one machine. The distributed version of Rendezvous is different in that it also requires the Rendezvous infrastructure on the view machines, and the view code is on the view machines instead of the central machine.
- **Weasel:** The model and view containers both require the Weasel infrastructure. The model code and any necessary external resources are only required on the model side, and the RVL runtime and the view code are only required on the view side. The model does not migrate.
- **Clock:** The model and view containers both require the Clock infrastructure. The model container also requires the model code and resources it needs, and the view container also requires the view code and access to an X server. The model does not migrate.

2.3.3 Performance

The literature for the above systems reports performance problems for all of the above systems. In fact, the broader literature consistently cites performance problems for centralized systems, due to latency (the time it takes for a message pair to travel from view to model and back) and bottlenecks on the network or at the centralized server. Bottlenecks are particularly troublesome for centralized synchronous distributed collaborative systems because multiple views are likely to need the same data at the same time.

Other reasons for performance problems in the above systems include:

- Inefficient client/server task splits (e.g., generating large views on the central system instead of the distributed peers) (XTV, Rendezvous),
- Inefficient protocols (e.g., using the *pull* vs. *push* MVC design pattern (see [GHJV95a])) (Weasel, Clock[GUN96b]), or
- Inefficient computations (e.g., re-computing views from scratch every time something changes) (Weasel).

Moving or replicating components (e.g., for mobility or latecomer support) can also be a source of inefficiency during the move or copy operation (XTV, Chung's infrastructure), though these operations are usually invoked in order to improve future interactive performance by locating components close to the participants who are using them.

Performance problems due to latency in centralized systems are often alleviated through the use of local view state. Unfortunately, in the systems described, this also leads to unshareable state, because it is difficult to share local state in centralized systems. Replication is also used in Rendezvous and Clock. In both systems consistency

problems are reported which either result in “erroneous user interface feedback” or “un-intuitive user interface behavior” (see [Urn98]). (In Rendezvous the replication is subtle. It involves replicating state variables in the model and multiple views, which are bound together by constraints. When two users simultaneously change one of these state variables users will at least see different sequences of events in their user interfaces, even if the views are guaranteed to be consistent when a quiescent state is reached.) Clock also implements locking, which can be inefficient. Course-grained locking can unnecessarily reduce concurrency, and fine-grained locking can be expensive in terms of the locking time required for multiple resources and the memory space required for fine-grained locks. In any case, consistency control mechanisms for replicated state involve either using a centralized component for serialization, or complicated algorithms for preventing or recovering from inconsistencies caused by inconsistent ordering of events to multiple replicas[GM94].

2.4 Contributions of this Work

I will now briefly describe the contributions of my work pertaining to the functionality, implementation, and performance issues raised about the centralized synchronous distributed collaborative systems described above, in order to motivate the focus of the analyses performed and conclusions derived in the following sections.

The major contributions of my work are as follows:

- I will motivate a particular variant of the MVC paradigm for use in centralized synchronous distributed collaborative systems, involving the use of an explicit controller and the push protocol. These are not new concepts, but the use of other variants has caused functionality and performance problems in previous systems as noted above.

- I will describe a novel solution to the classical tension between consistency with bottlenecks and high latency on the one hand and responsiveness with sharing and concurrency control problems on the other, in highly interactive centralized synchronous collaborative systems. My solution is based on the identification of classes of model entities⁴ that can be migrated automatically and extremely quickly, while maintaining most of the simplicity of centralized semantics and implementations. My solution will result in systems nearly as responsive as replicated systems in most cases, without the complicated implementations and potential un-intuitive user feedback of existing replicated systems.
- Based on an analysis of the view computation components in the above systems, I will identify a class of view computation components that can migrate quickly and easily, while enhancing the determinism of the system.
- I will identify a means of classifying and organizing model state in a manner that allows the infrastructure to better manage the various classes of such state. In particular, this classification and organization will help the infrastructure to enhance performance, reduce user interface and collaboration awareness in applications, increase desirable view divergence possibilities, and present divergence opportunities to users in an understandable manner.

⁴In this paper, I will use the term *entity* where one might expect the word *object*. I have purposely avoided the word *object* because of its common definition in object-oriented systems, where it implies support for the fundamental object-oriented concepts of abstraction, encapsulation, polymorphism, and inheritance. *Entities* in this paper are such things as applications, processes, code, object-oriented objects, data structures, files, databases, devices, mathematical functions, and concepts, which may or may not meet object-oriented requirements.

2.5 Analysis Framework for Centralized Synchronous Distributed Collaborative Systems

I will now present an analysis framework that can be used both to analyze centralized synchronous distributed collaborative systems like those described above, and to point the way to improved architectures addressing many of the above concerns. The analysis framework that I am proposing will identify various entity classes in the systems being analyzed along with various attributes of each class.

The entity classes identified in the above systems are shown in Figure 2.9, while the attributes of these entity classes that I will be discussing are shown in Figure 2.10.

- Models:
 - Process-Based
 - Object-Based
- Protocol Manipulators
- View Computation Engines:
 - State-Machine-Based
 - Constraint-Based
 - Function-Based
- Local View State Repositories
- View Specifications:
 - Imperative
 - Declarative
- View Realizations
- Controllers

Figure 2.9: Entity Classes

I will discuss each entity class in turn, covering each of the attributes for each of the

- General attributes:
 - The type of the entity.
 - The representation of the entity.
 - The role the entity plays in the system.
- The divergence possibilities of the entity in the system:
 - actual
 - desired
- The visibility of the entity:
 - to developers
 - to the infrastructure
 - to users
- The location of the entity in the system:
 - actual
 - desired
- The mobility and ease of mobility of the entity in the system:
 - actual
 - desired

Figure 2.10: Entity Attributes

above systems.

2.5.1 Models

Models are entities that play the role of the MVC model by maintaining the primary semantic state for the system. In Chung's infrastructure, the kind of data represented in the model is dependent on the layers between which logging is im-

plemented and the details of the system being logged. The semantics of the state are largely invisible to the infrastructure, but this is dependent upon the amount of semantic information provided by the loggable. In all the other systems described above, the model mixes state for both the application domain and the user interface domain. For example, in the Clock example of Figure 2.7, the Depressed ADT represents user interface domain state, while the Toggle ADT represents application domain state. While these are separate ADTs, the infrastructure cannot identify their different roles. In XTV it is impossible to distinguish between application and user interface state, and in the other systems, any such distinction is application dependent and invisible to the infrastructure.

Model state in the above systems is represented either as a process or as an object structure. In XTV the model is represented as a process. In Rendezvous, Weasel, and Clock it is represented as object structures, and in Chung's infrastructure either process or object representation is supported. Process representations are typically black box state machines. Object representations have the advantage of giving the infrastructure finer-grained access to model data, making it possible to attach constraints to objects or to use objects or object method calls as function parameters.

Process representations force coarse-grained divergence possibilities, as in XTV, where the entire application must be shared wholesale. Object representations enable finer-grained divergence, as in Rendezvous or Weasel. Chung's infrastructure does not support fine-grained divergence even when the model is object-based, because it does not know enough about the semantics of the objects whose protocols are being logged. Clock does not support divergence because its ADTs are strongly tied to a particular user interface structure.

In all of the above systems, the model structure is visible to developers either at the course-grained process level or at a fine-grained object level, so that the developer can

implement divergence at whatever granularity the model provides. XTV and Chung’s infrastructure expose process-level models to the user for coarse-grained divergence at the application level, while Rendezvous and Weasel do the same with entire model object structures. With all these systems except XTV, models can be associated with different views. In Clock the individual ADTs are distributed throughout the architecture, whose structure is determined by the user interface. Thus, as with XTV, models are tightly bound to user interfaces. All of these systems provide only course-grained, entire-model visibility to users.

Ideally, model state would be object-based with a separation between application and user interface domains, and object granularity would be at an appropriate level to represent units of divergence (neither too high, obscuring finer-grained divergence users might want, nor too low, making divergence too detailed and complicated). Objects would be visible to both developers and users so that they could both control divergence scenarios, and to the infrastructure, so that it could implement automatic performance enhancements and expose divergence opportunities to developers and users.

Undifferentiated models must in general be located where they have all the necessary container support and access to external resources. Container support for models is particularly demanding, since models are typically responsible for observing and manipulating the environment around them, and their container and external resource needs are application dependent. Such dependencies are typically invisible to the infrastructure. This is true of all the systems described above.

Model location would ideally be as near the currently-interacting participant(s) as possible, to reduce latency. The membership of this set of participants is highly dynamic. XTV and Chung’s infrastructure are the only systems described that support dynamic model mobility. Even in these cases, such mobility is non-deterministic, and

takes on the order of seconds. Ideally, such migration would be in the 100 millisecond range or faster, to avoid disrupting the work of participants[Shn98]. The demands for container and external resource availability on the one hand, and for low latency on the other create a tension between required and desired model location that is a central focus of this paper.

2.5.2 Protocol Manipulators

Protocol manipulators like Chung’s loggable and XTV’s pseudo-server are infrastructure components typically used to add collaborative capabilities to existing single user applications. They can be represented either as independent processes or as embeddable software components. They are typically not visible to developers, because the whole point is to add their functionality transparently to applications. They are often made visible to users through a session user interface so that users can create and manage sessions and attributes of sessions such as floor control. Whatever divergence the infrastructure may be able to support is exposed via this user interface.

Protocol manipulators are interposed between layered components of existing software systems. Given network communication, they can often be located anywhere, but they are usually located centrally near the model or in a distributed fashion near each view. If they are embedded, of course, they are located with the component in which they are embedded. Since they are infrastructure, which is under the control of the infrastructure developer, they can usually migrate fairly easily with either the model or the view.

2.5.3 View Computation Engines

The purpose of a view computation engine is to create a mapping between a model and a view specification, where the view specification defines the current state of a user interface. View computations are represented in the above systems by state machines, constraint bundles, and functions. XTV and Chung’s infrastructure represent these engines as state machines, Rendezvous represents them as links (constraint bundles), and Weasel and Clock represent them as functions.

Ideally, view computation engines should be distributed and located as near participants as possible. There are several reasons for this location preference:

- Models are usually smaller than views, and changes to models are usually smaller than changes to views. Locating views near the participant therefore reduces network traffic, increasing performance and scalability.
- Performing view computations on participant hosts distributes the computation and reduces the bottleneck at the central location. This in turn increases performance and scalability. Distribution of view computation is particularly important when the views differ and must be computed separately.
- In the other direction, lower-level (e.g., user-interface-level) events are usually more frequent than higher-level (e.g., model-level) events. Locating view computation on the participant’s side of the network therefore reduces network traffic and the frequency of over-the-network latency delays.

XTV, Rendezvous, and, in some cases, Chung’s infrastructure and Clock generate views on the model side of the network. Weasel and the experimental distributed Rendezvous are superior in that they compute the view on the participant side of the network infrastructure, and Chung’s infrastructure and Clock are capable of doing

the same.

Locating a view near a participant implies migrating that view when the participant moves, if user mobility is to be supported. Migrating a view is essentially the same as supporting latecomers, except that any view-specific state must be migrated as well. Migrating view computations is typically much easier than migrating models because in general, views have more restricted and better defined semantics than models. All of the described systems support view migration or latecomers. Since XTV and Chung’s infrastructure must migrate state machines, their task is the most difficult, and is in general non-deterministic, as described earlier. Migrating views with Rendezvous is easier because view computation is constraint-based, and views can therefore be re-computed automatically from model state. Migrating views with Weasel and Clock somewhat easier still, since the parameters to functional view computation can be at a higher level of abstraction than Rendezvous constraints and are therefore easier to establish. Unfortunately, Rendezvous, Weasel, and (perhaps) Clock allow local view state for performance reasons, and the papers on these systems do not indicate that they provide any mechanism for migrating that state.

In all cases, view computations are visible to developers, since the code to compute them is the developer’s responsibility. View computations are also visible to infrastructures, except in the case of XTV and (sometimes) Chung’s infrastructure. Making view computation visible to the infrastructure opens up the possibility of allowing the infrastructure to position the view computation components in the network. Rendezvous and Weasel can presumably make view computations available to users so that they can mix them with different models, although this is not explicitly discussed in the literature. With Clock and XTV, view computations are tightly bound to models, making such recombination impossible, and with Chung’s infrastructure the ability to recombine models and views depends on the system being

logged.

2.5.4 Local View State Repositories

Local view state repositories are facilities for maintaining inputs to view computations that are local to each participant and can be manipulated by the local controller or view. They function as a means of providing for private (unshared) data and are also used to alleviate much of the latency in centralized systems. Unfortunately, the distinction between private and shared data is often dynamic, and local data is difficult to share in a centralized system. There is therefore a tension between the desire to locate view-specific data near the participant, where it results in greater responsiveness, and locating it near the central model, where it can more easily be shared. As is true with model state, alleviating this tension is a major focus of this work.

XTV does not maintain any local view-specific state (other than the user interface itself, which is not being considered here), and Chung’s infrastructure may or may not, depending on the logged system. Rendezvous allows for view state that is not bound by or to model state via constraints, and is therefore local state. Weasel explicitly supports local view state (and Clock may do so). None of the above-mentioned systems advertise that they support view state migration as users migrate. Ideally, any local view state should be both shareable and migratable, as well as capable of providing good responsiveness.

Local view state is typically visible to developers and the infrastructure. It is not usually visible as such to users, as the location of state is an implementation detail.

2.5.5 View Specifications

View specifications determine what a user interface should look like. They can take the form of a sequence of imperative commands (as with the X Window System protocol), or a declarative representation of the current state of the user interface that can be translated to window system commands.

XTV and Chung’s infrastructure take the imperative approach, while Rendezvous, Weasel, and Clock take the declarative approach. Declarative representations of the user interface are easier for the view computation to generate and keep up to date, as it is dealing with a static representation of the current state rather than a state machine. That is, a declarative view specification isolates all of the difficult work of dealing with the window system state machine and allows the infrastructure to perform that function. Declarative representations also allow efficient snapshots of the user interface to be easily obtained, and they eliminate the potential of not being able to retrieve the entire user interface state. They are also useful to the developer for debugging and testing.

View specifications are visible to the developer and the infrastructure, but not typically to the user. Making a declarative representation available to the user enables him to take user interface snapshots in a structured (i.e., analyzable), efficient, and easily rendered form.

View specifications, like view computations, should be near each participant, and should migrate with the participant if user mobility is supported. In the case of state-machine view computations, a declarative view specification could simplify and speed migration, as the user interface could be automatically regenerated from the specification. In the case of constraint-based or function-based view computations, it is not necessary to migrate the declarative view specification, as it will be regenerated

automatically.

By the time a view specification is generated, it is too late to think about divergence; any divergence among users is already represented in the view specification.

2.5.6 View Realizations

A view realization is the actual user interface generated from a view specification, via a window system or user interface toolkit. View realization must be located with each participant, and must migrate with the participant if user mobility is supported. XTV, Rendezvous, and Clock realize their user interfaces using the X Window System. Chung's infrastructure can support different window systems. It is unclear from the literature what window system was used by Weasel.

The view realization is visible to the infrastructure and, of course, to the user. If the view specification is imperative, the view realization is visible to the developer; if the specification is declarative, it is not. Like the view specification, the view realization reflects any per-view divergence.

2.5.7 Controllers

The job of the controller is to map low-level events in the user interface domain to high-level operations in the application domain. Of the systems described, only Clock supports this controller function. XTV, Rendezvous, and Weasel typically operate on models at a level of abstraction lower than the model domain. Chung's infrastructure just logs and replays operations without understanding their domain. Controllers are typically represented as event handlers for user-interface domain events. They should be located near the participant for the performance and scalability reasons described earlier.

When present, controllers are visible to developers and the infrastructure. They can be made visible to users in the form of a choice among user interface bindings (e.g., emacs vs. vi editor bindings), which can often be customized per-user. Controllers thus represent a divergence mechanism that works in the input direction. When controllers are present and when they allow per-view customization, they must migrate with the user. Such migration is typically easy, because user interface bindings are usually declarative and are visible to the infrastructure and developer, and because no local execution state need be maintained between user interface events⁵.

2.5.8 Analysis Summary

Based on my analysis of XTV, Chung’s logging infrastructure, Rendezvous, Weasel, and Clock I have observed the following about centralized synchronous distributed collaborative systems:

- Centralized systems suffer from performance and scalability problems because of latency between distributed and centralized locations, and because of bottlenecks on the network and at the central location.
- Models are difficult to migrate in general, because they are state machines and because of their container and resource access requirements. Classification of models by their properties can enable the infrastructure to support various classes differently. This may allow us to break down the model migration problem such that we can deal with it more effectively.
- In order to be able to manage divergence from WYSIWIS sharing, potential and useful divergence points in models need to be exposed to users in an un-

⁵Incomplete gestures, which may require some local controller state in-between user interface events, do not need to be carried across a user migration.

derstandable manner. Model classification may help in this regard as well.

- Model organization should be independent of user interface structure, so that models can be reused with radically different user interfaces.
- Views should be separated from models and located near participants, for performance and scalability. They should migrate with participants.
- Reducing container and external resource requirements facilitates migration of entities.
- Functions are easier to migrate than constraints, which are easier to migrate than state machines.
- Local view state should ideally be separated from distributed view computations and centralized so that it can be shared, and so that views can be migrated more easily. On the other hand, local view state mitigates the latency problems of centralized architectures.
- Representing view specifications declaratively instead of imperatively simplifies view computation.
- Maintaining a separate controller component, as in the original MVC paradigm, helps to separate models from views, which in turn facilitates component reuse and increased divergence possibilities. It also simplifies concurrency control and helps to remove user interface and collaboration awareness from models.

It is clear from these observations that most of the performance, implementation, and functionality problems of existing centralized synchronous distributed collaborative systems stem from one or more of the following:

- Not maintaining a controller separate from the view in an MVC-like architecture, and using MVC inefficiently.
- Lack of mechanisms to classify entities by their properties and expose these classifications to the infrastructure, developer, and user.
- Issues surrounding the locations, replication, and migration of entities.

The first issue is mostly independent of the others, and the solution is straightforward: use the *push* MVC paradigm, and implement a separate controller component that maps between user interface and application domain operations. The second and third issues are inter-related because entity classification can help us resolve location issues. In the next chapter I will propose an architectural philosophy for the classification of entities by properties such as their role in the system, their container and external resource requirements, and their desired location and migration characteristics.

2.6 Other Related Work

2.6.1 Perspective-Like Constructs and User Models

The concept of Concur perspectives, which are introduced in Chapter 3, was heavily influenced by a paper by Schuckmann, et. al. [SSS99], in which they propose the following:

- Separation of models pertaining to the problem domain from those pertaining to the user interface, in order to promote reuse of both.
- Shareability of *all* user interface models, to enable a wide range of dynamically selectable coupling/divergence scenarios, and to promote group awareness.

This paper also promotes separately maintained but linked *user models* in the UI and problem domains. In this context, a user model is a model that maintains information about the users interacting with a particular UI, or information about users that is part of the problem domain.

While this paper promotes underlying, Concur-like mechanisms for enabling a wide range of coupling and divergence scenarios, it does not propose any particular mechanism for managing coupling and divergence.

2.6.2 Coupling Systems

Suite[DC95] represents the most thorough analysis of coupling and divergence possibilities in the literature. Suite automatically generates data-specific user interfaces (editors) from arbitrary application data structure specifications. It distributes these editors to multiple users, associates them with a centralized application that maintains committed values of the data structures, and manages couplings among data items in related editors. Each data item is presented independently in the UI. Suite's focus is on *when* sharing of individual data items occurs.

Each editor maintains its own copy of each data item, which can be edited independently. Couplings specify, for each data item, when changes should be communicated to and received from other editors of the same data item. For example, a particular item can be transmitted on every incremental change, only when it is syntactically or semantically correct, on time intervals, or when explicitly committed or transmitted by the user. Coupling parameters can be specified for each pair of users, and a data item must satisfy both transmission and reception specifications in order to be transferred between a pair of users. Coupling parameters can be specified as Suite defaults, by the application programmer, or by the individual users. A coupling

editor is provided for user specification of coupling parameters, but the user must make the association (by name) between a data item and its representation in the editor. Because there may be many data items, Suite provides various grouping and inheritance mechanisms based on the underlying data structures, to simplify coupling specifications.

The full generality of Suite is applied only to model (semantic) data, not user interface coupling. Particular aspects of the user interface (e.g., scrolling, selection, and formatting of data) can be coupled, but this a boolean specification for each aspect (it's shared exactly or it isn't at all), and the set of shareable UI aspects is fixed.

Suite is built on the notion of simultaneously maintaining various divergent versions of the same object and transferring values among these versions, based on coupling parameters. Thus, Suite provides for a wide (but fixed) range of coupling scenarios for each of a number of directly-presented data items.

While Suite's generation of user interfaces and its automatic⁶ management of coupling scenarios among editors off-loads much work from application developers, the consequence is that it is not as general with respect to the possible set of user interfaces and divergence scenarios (in terms of the *what* of coupling and divergence) as Concur. In Concur, view code determines the set of divergence possibilities by the way in which it specifies its input parameters, and it determines how each potentially divergent data item affects the UI. The Suite and Concur methodologies represent different points in the collaboration infrastructure space, each suitable for its own intended use: Suite provides more automation and less generality, while Concur has the opposite characteristics.

⁶Suite does offer callbacks and methods that allow applications to implement their own arbitrary coupling semantics, at the cost of automation.

2.6.3 State-Management Systems

In this section I will discuss what I will call state-management systems. These are systems whose purpose it is to replicate state among components by extracting state from one component and reproducing it in a replica.

Roussev’s Programming Patterns [Rou03] focuses on the issue of abstraction flexibility, the ability to share a wide range of programmer defined abstractions. He demonstrates that the logical structure of a program can be dynamically determined and the state of its objects accessed by identifying common method signature patterns. A small set of patterns can often be used to identify the methods for reading, writing, and modifying the state of a large number of objects. Programmer-defined handlers can then be provided to map from generic operations understood by the collaborative infrastructure (e.g., to read, write, or compare state) to the corresponding pattern-specific operations. These generic operations can then be used to access and communicate (e.g., replicate) the state of the objects. An event mechanism is required to determine when and how objects have changed state. If the objects already supply such an event mechanism, it can be mapped to the infrastructure’s generic event mechanism. Otherwise, through minimal code changes to the application, events can be added that essentially encode state-changing method calls so that they can be communicated to the infrastructure.

Roussev’s work also provides the infrastructure for Suite-like coupling of state among related objects (e.g., replicas). Thus, coupling specifications can be used to determine when state changes are communicated among objects⁷. In Roussev’s work, however, the objects are arbitrary; they are not the automatically generated editor UIs of Suite. Thus, Roussev’s infrastructure can be used in the context of arbitrary

⁷Roussev’s thesis did not address concurrency problems that arise when objects are modified simultaneously by multiple users.

user interfaces.

Roussev's programming patterns were implemented in Java using its reflection capabilities, so it can be applied to compiled code, assuming that adequate event mechanisms are already available in that code. His methodology could be applied to any programming language supporting reflection.

Chung's log-based collaborative infrastructure [Chu02] takes a different approach to state management. Chung's approach is to establish replica state by logging and replaying communications between the layers of a layered application. His infrastructure supports state establishment by both direct state transfer (and update) and command sequences.

Chung's infrastructure can be used to replicate components of any of the layers of an application (e.g., model, view, window, or screen). It can therefore be used to implement centralized or replicated architectures or hybrid architectures with both centralized and replicated sub-architectures. It also supports dynamic transitions among these architectures, by which the collaborative infrastructure can adapt to dynamic changes in the structure, needs, and performance characteristics of a collaborative session.

One of Chung's goals was to be able to add collaborative capabilities to as wide a range of applications as possible, including existing single-user applications. To accomplish this, he requires inter-layer protocols to be mapped to an abstract I/O protocol specified by his infrastructure. Another of his goals was that his infrastructure be as unaware of the semantics of the application protocols as possible. This goal is in conflict with the need to manage non-determinism and non-idempotent outputs and to reduce command sequence lengths. Thus, for correctness and efficiency, the abstract I/O protocol contains facilities for tagging communications with semantic information. Chung uses heuristic techniques such as message counting to determine

when a replica is in the desired state. However, given the typical scenario of existing applications with incompletely-specified semantics and uncooperative protocols, there usually remains some uncertainty with respect to the ability to deterministically put components into the correct state without duplication of non-idempotent outputs, and to bound the command sequence length. This is especially true when replicating the model layer for a replicated architecture. Once a component is in an incorrect state, there is often no way to ensure that future states are correct. While satisfactory results can usually be obtained with enough experimentation and tweaking of the protocol mapping, this is an arduous process. Chung's dissertation is a good exposition on what can go wrong, and how to address the issues when you have no other choice.

By and large, it is much easier to set the state of a view than a model. Views have a specific, limited purpose – to define the user interface of an application in terms of some standard UI technology. The size of view state is bounded in practice by this limited purpose. There is also little reason for extraneous inputs, since the user interface should ideally be deterministically defined by the application. When such inputs exist (e.g. different font sets for different instantiations of a window system), they are usually inputs to the standard underlying UI technology, a well-defined sub-component for which work-arounds can most often be found. Such work-arounds can be reused from view to view. It is not the purpose of a view to affect its external environment, other than to produce transient user interfaces on a display used for user/machine communication, so there is little reason for a view state machine to emit non-idempotent outputs external to the display. Finally, because of its specific, limited purpose, view code can often be easily transported and executed in limited, replicated contexts.

Models, by contrast, are arbitrary, application-dependent components. Since their

purpose varies from application to application, their state size is not bounded. They often have context-specific inputs (e.g., file systems and environment variables, represented by Resources in Figure 2.11) that may affect their state transitions in non-deterministic ways from context to context. Models are also often required to emit non-idempotent outputs to affect their environments in various ways. In contrast to view code, model code is generally more difficult to export and execute in a limited, replicated context, because it may have arbitrary dependencies on resources and infrastructure.

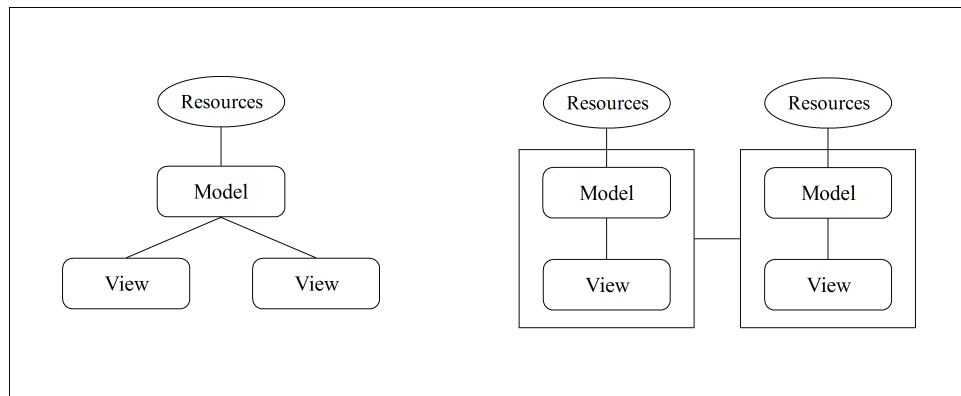


Figure 2.11: Centralized and Replicated Architectures

It is largely because of the state-management difficulties discussed in this section that I have chosen a centralized architecture and functional views for Concur. The centralized architecture eliminates the need to replicate semantic models. Functional views simplify state management and facilitate determinism and avoidance of non-idempotent outputs (side effects, in functional language terminology).

Chapter 3

Entity Taxonomy

3.1 Introduction

In this section I will first discuss the distinction between model and view state implied by the MVC paradigm. This discussion will have a unifying effect, suggesting that model and view state should be handled similarly. (Since I will be advocating treating view state as a model, I will use the terms *user interface (UI) domain state* for view state and *application domain state* for model state, to avoid confusion.) Then I will introduce a different classification of the unified state, based on properties such as entity roles, container and resource requirements, and desired location and migration characteristics. The end result will be an entity taxonomy that will point the way to enabling most state to be efficiently shared and a wide range of divergence possibilities to be exposed to users in an understandable manner.

3.2 Application Domain vs. UI Domain State

Schuckmann, et. al. at GMD[SSS99] propose a shared object model which is a foundation for my work on entity classification¹. They begin by proposing the following two policies regarding the modeling of shared objects:

- *Separation of application domain and UI domain state.* State in the user interface domain (e.g., scroll bar positions and selections) should be separated from state in the application domain. This policy, essentially the same as that of dialogue independence[HBR⁺94], facilitates multiple views of the same model and reuse of both models and views. (This policy may seem obvious for any architecture using the very common model/view separation advocated by dialogue independence, but it takes on new significance when the UI state is separated from the user interface itself, as recommended by the second policy below.)
- *Shareability of all state.* UI state should be shareable, just as application state is. Making UI state shareable implies replicating and synchronizing it or separating it from the (typically distributed) view computation component and centralizing it. This policy supports the sharing of parts of the user interface itself. Unfortunately, this usually comes at a cost in performance or in the complexity of UI state consistency management.

The rightmost diagram of Figure 3.1 illustrates these aspects of the GMD shared object model (which I will henceforth call GMD). GMD essentially advocates a “separate but equal” policy for application and UI state. Another way of saying this is that there should be two model domains: one for the user interface and one for the

¹Schuckmann, et. al. use the term *application* to refer to the user interface, while I use the same term to refer to the code implementing model behavior. In my discussion of GMD’s work, I have therefore used my own terminology instead of theirs, in order to avoid confusion.

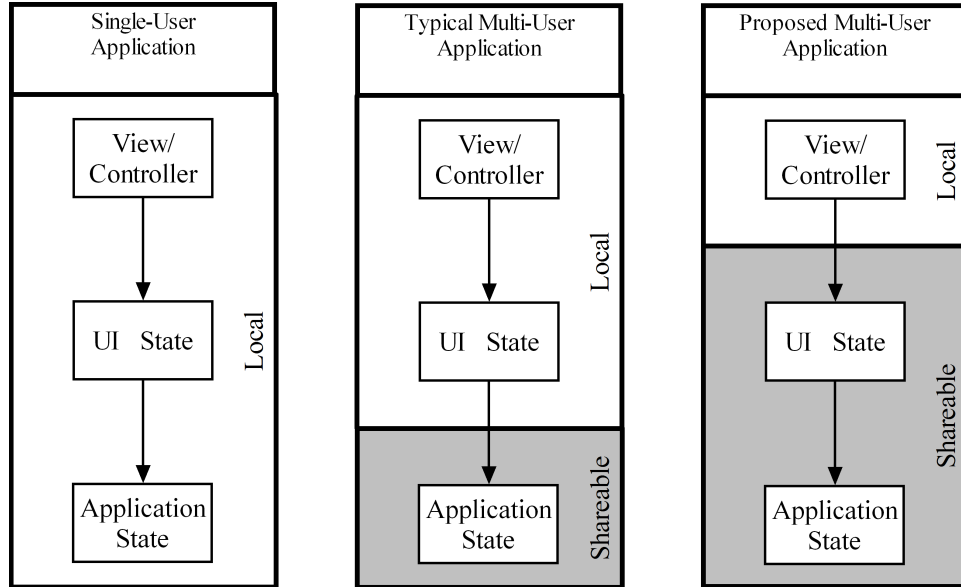


Figure 3.1: GMD Shared Object Model

application. Representations of both are intermixed in the view.

Based on the discussion so far, GMD only supports strict WYSIWIS sharing and sharing of the application state using independent views (Figure 3.2). To see how GMD achieves finer-grained sharing (relaxed WYSIWIS), we must first look at its UI state representation in more detail. Consider the sample Notepad application of Figure 3.3. GMD would represent the UI state of this Notepad as shown in Figure 3.4. Like a Clock architecture, this tree mimics the widget hierarchy in the user interface, but unlike Clock it contains only UI state, not application state. It can therefore be thought of as a model for the user interface. Since view sharing is accomplished by sharing the same UI state tree, the entire state of the user interface is shared WYSIWIS by default.

In order to support relaxed WYSIWIS (e.g., independent scroll bars and selections), GMD uses what I will call a *conditional data structure* (Figure 3.5). That is, nodes in the tree can present multiple alternatives. A particular view selects one

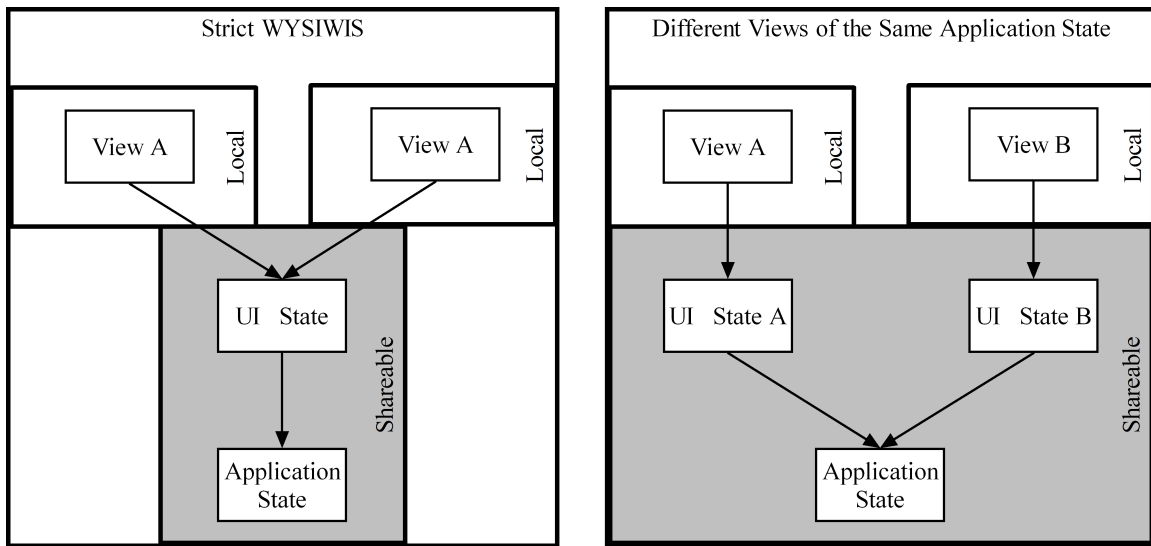


Figure 3.2: GMD Coarse-Grained Sharing

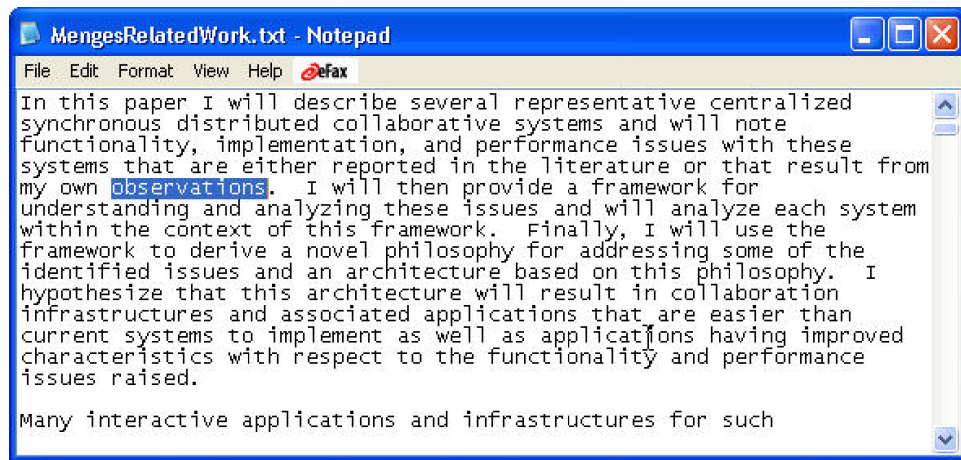


Figure 3.3: Notepad

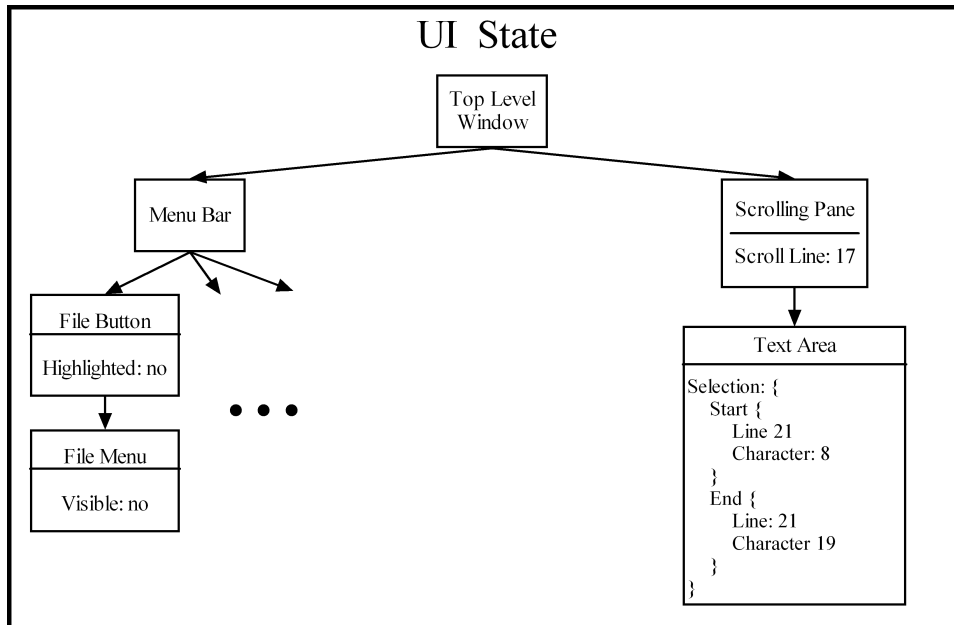


Figure 3.4: GMD Notebook UI State

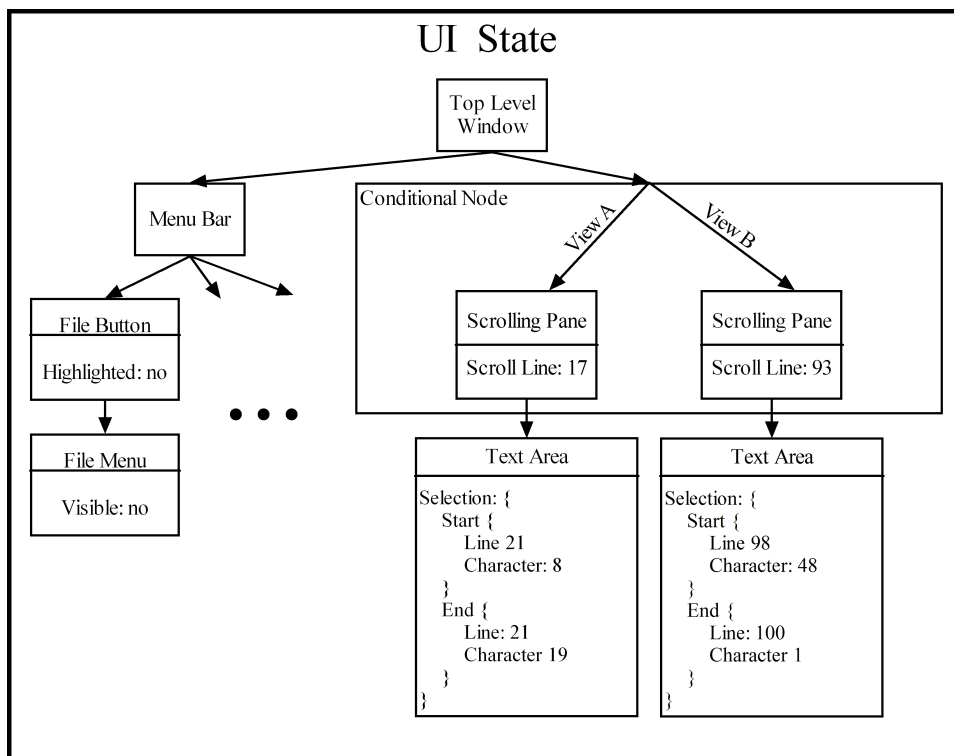


Figure 3.5: GMD Conditional Data Structure

alternative based on the identity of the view. In the figure, the scroll bar and selection state diverge, while the menu state remains coupled.

Unfortunately, UI state trees with conditional nodes can get complicated if you are trying to support a wide range of fine-grained divergence units. Also, as in Rendezvous and Clock, the GMD tree-structured model for the user interface is compelled to closely follow the component hierarchy in the view realization. This binds the UI state model closely with the user interface, complicating the coupling of UI state in views that may be very different.

3.3 Development of a Unified Model

Starting with the GMD model, I will now develop a line of reasoning that results in a unified model, which will later be re-classified along different lines. First, I propose a reorganization of the UI state such that it really does have a “separate but equal” standing with application state (Figure 3.6). Note that the view directly

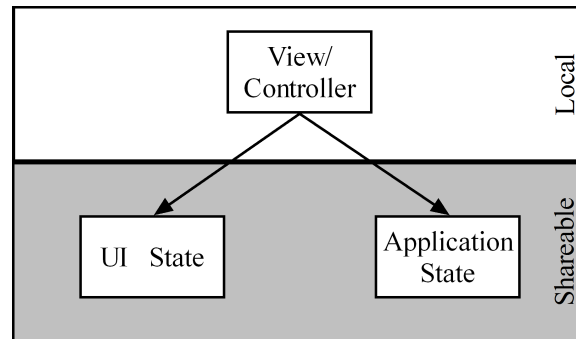


Figure 3.6: View and Model State Equality

accesses both UI and application state, instead of accessing model state through the UI state component, as was the case in Figure 3.1. Second, since the UI state should not conform to the user interface component hierarchy, I will flatten the UI

state (Figure 3.7). I will assume, as this figure shows, that the application state

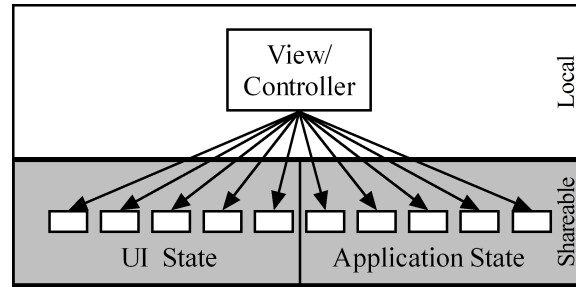


Figure 3.7: Flattened State Organization

is flat as well. These figures show that both UI and application state should be composed of smaller units (as they already were in the GMD model), which facilitates individual coupling or decoupling of smaller units of state (Figure 3.8). This, in

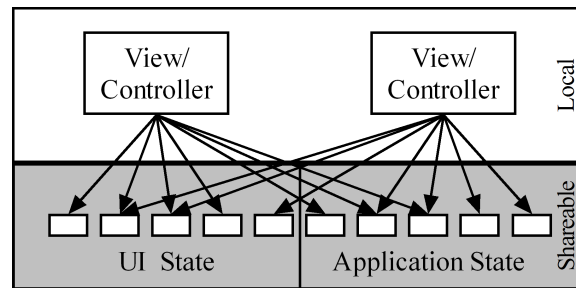


Figure 3.8: Coupled and Divergent State

turn, dramatically increases the potential set of divergence scenarios that can be supported. In fact, I am proposing that the units of state be organized such that they represent exactly the set of useful units of divergence. Finally, in Figure 3.9, I will remove the distinction between UI and application state in preparation for deriving a new classification relating to role, container and resource requirements, location, and mobility.

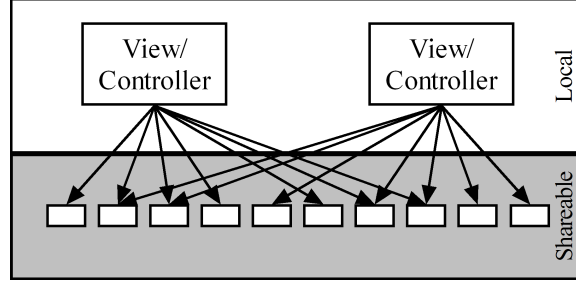


Figure 3.9: Unified Model State

3.4 A State Classification Based on Entity Properties

This section derives a state classification based the roles, container requirements, desired location, and desired mobility of entities.

Synchronous distributed collaboration invariably requires the replication of entities. This is clear from Dewan’s generic collaboration architecture[Dew99] (Figure 3.10), where at least one, and typically more than one of the bottom-most layers is replicated in some fashion. Whether collaboration is WYSIWIS or not, some entity or entities at some level of abstraction must be replicated in order for collaboration (sharing) to take place. Support for latecomers and user mobility further require such entities to be mobile, that is, to be capable of being moved or cloned from one environment to another.

Entity mobility infrastructures can be complex. Simplifying the development of entities that are to operate within such an infrastructure usually requires *transparent* entity mobility, i.e., the ability to move entities from one environment to another without the knowledge or participation of the entities being moved or of other entities that know about them². Implementing a transparent mobility infrastructure,

²Some entity management systems are purposely non-transparent so that entities can adapt to

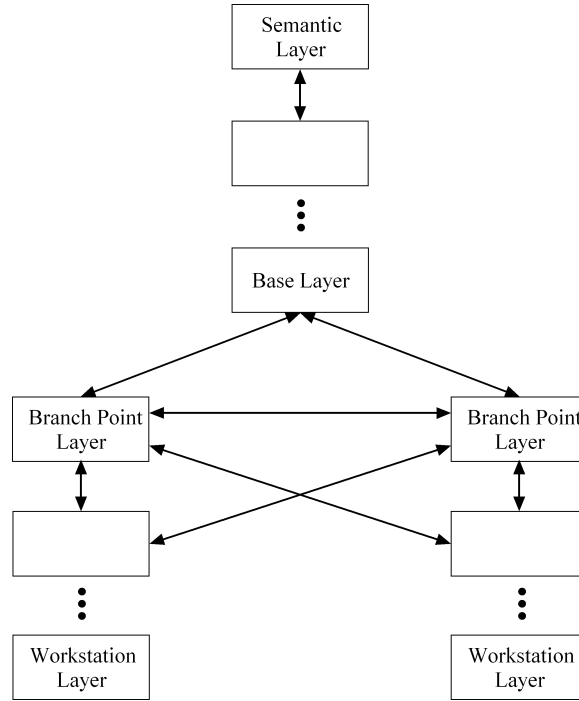


Figure 3.10: Dewan's Generic Collaboration Architecture

however, can be difficult, for the following reasons:

- Suitable and identical entity container technology must be replicated at all execution environments that might host a given entity.
- Any code associated with the entity (e.g., for object behavior) must already be available at the target environment, or must be replicated there as part of the move.
- The state of the entity and any processes or threads associated with it must be captured at the source and reproduced at the target of the move.

their current execution environment. I will only consider transparent systems here, since the main purpose of synchronous collaboration is to share things to different locations, not to adapt to differences at these locations. Any such adaptations that may be required in synchronous collaborative systems can usually be done transparently, so that the entities themselves do not need to consider mobility.

- Any external entities referenced by a relocated entity (e.g., files, databases, and other system resources or other mobile entities) must either be moved or replicated with the relocated entity, or accessed over the network using a location-transparent reference.
- Any references to a relocated entity must either be location-transparent or must somehow be redirected to the relocated entity's new location.
- Any information an entity might receive from referenced entities (e.g., via queries or events) must be consistent over time, whether or not either or both entities have moved during the interim.
- Various performance and resource utilization issues must be considered, such as the time and network bandwidth costs for moving entities and for accessing non-local references.
- Mobility management policies must be implemented which determine how to map entities to execution environments and how and when to change the mapping dynamically as situations change.

The existing entity mobility literature generally focuses on *process* migration (e.g., Berlix[Lux95] and Chung's logging infrastructure[CD96]), (object-oriented) *object* migration (e.g., COOL[AJJ⁺92] and Emerald[SJ95]), or mobile agents (e.g., Mole[BHR97] and MOA[MLC98]). Many of these systems take the approach of trying to solve all of the difficult problems by providing transparent migration to complex objects. As a result, container requirements are typically demanding and migration times large, on the order of seconds. In order to support non-disruptive migration in synchronous collaborative settings, we need migration times below 100 milliseconds. Thus, I take the opposite approach, focusing on how to minimize the complexity of

migrated entities in order to make migration simple and fast. My migratable entities are sub-objects – things that have less than full object-oriented properties.

What are the characteristic properties that would make entities easy and fast to migrate? Here are the most important ones:

- Minimal, standard, technology-independent container requirements.
- Either no external references to other entities, or only location-transparent references.
- Small size.
- Lack of execution (thread or process) state.
- Immutability.
- Lack of non-transient state.
- Requiring only generic (as opposed to application-specific) operations on state.
- Invisibility to the application.

It is not necessary for entities to have all of the above properties in order to achieve fast and simple migration³, but all of these properties are potential means toward that end. Table 3.1 summarizes the entity classes I have identified, along with their properties. In this table, attribute values violating one of the above criteria for fast and easy mobility are underlined. I will now discuss each of these entity classes in turn.

³In fact, an entity meeting all of these properties might not be very useful.

Entity	Container Requirements	External References	Size	Execution State	Mutable	Non-Transient State	Non-Generic Operations	Visible to Application
View Computation Function	portable language, standard data representation	location-transparent	<u>moderate</u>	no	no	no	N/A	no
Controller	<u>portable language</u> , standard data representation	location-transparent	<u>moderate</u>	no	no	no	N/A	no
Data Perspective	standard data representation	none or location-transparent	small	no	<u>yes</u>	<u>yes</u>	no	no
Timer Perspective	<u>clock</u> , standard data representation	none	small	no	<u>yes</u>	<u>yes</u>	no	<u>yes</u>
Mobile Model	standard data representation	none or location-transparent	small	no	<u>yes</u>	<u>yes</u>	no	<u>yes</u>
Immutable Model	standard data representation	none or location-transparent	<u>large</u>	no	no	no	no	<u>yes</u>
Immobile Model	arbitrary	arbitrary	<u>large</u>	<u>yes</u>	<u>yes</u>	<u>yes</u>	<u>yes</u>	<u>yes</u>

Table 3.1: Entity Classes

3.4.1 View Computation Function

A purely functional view computation written in a portable language is extremely easy to migrate. It makes few container demands, as it only needs a runtime environment for the portable language and access to a standard data representation facility for its inputs and outputs. (The inputs are essentially the entity classes from Table 3.1, all of which can be represented using the standard data representation facility. The output is a view specification, which can be represented using the same facility.) The view computation function maintains no non-transient state (i.e., state that cannot be easily re-generated from centralized state), and execution state does not need to be maintained because the purpose of the function is to generate its current output based only on its current inputs. Its external references are to its input parameters and output, which can be referenced in a location-transparent manner. The function is immutable and invisible to the application, further facilitating migration. View computation functions will need to be migrated to support latecomers and mobile users.

3.4.2 Controller

In Table 3.1, a controller has exactly the same properties as a view computation function. It runs in the same container environment as the view function and has the same needs for a portable language and standard data representation. Its external references are to the view specification produced by the view function (for user-interface event inputs) and a controller data structure (for invoking operations in the application domain) represented using the standard data representation facility. Both of these can be referenced in a location-transparent manner. The operations invoked by all participant controllers are aggregated and applied to a common centralized

structure through which they are made available to the application. The controller does not need to maintain non-transient controller state or execution state (because execution state does not exist between events), it is immutable, and it is invisible to the application. Thus, a controller is as easy to migrate as a view computation function. Controllers need to be migrated along with their associated view computation function.

3.4.3 Data Perspective

A data perspective is a small data item typically used to represent UI state. It has no behavior other than the representation of simple state, so all its operations are generic data manipulation operations, and no execution state need be maintained. This means that it can be represented using a standard data representation facility. It may reference other perspectives or models, but these references can be location-transparent. A data perspective is mutable and it maintains non-transient state (i.e., unreproducible state that needs to be migrated when a user moves), so support for state migration is required. Data perspectives are distinguished from mobile models in that they are not visible to applications and need not be persisted beyond the lifetime of the session.

Data perspectives have many purposes:

- They can be used, as their name implies, to provide some perspective on model data. For example, they may be used to represent scroll bar positions or a viewpoint in space for a 3D model.
- More generally, they can be used to represent the state of the user interface (as opposed to the state of the application). For example, a data perspective can be used to represent a mouse cursor or telepointer position, whether or

not a drop-down menu is visible, and the highlighted state of menu items and buttons.

- If a view computation function is able to compute multiple styles of views of the same data (e.g., a pie chart vs. a bar chart, or an analog clock vs. a digital one), a data perspective can be used to choose among these view styles. Similarly, they can be used to select overlaid viewing options, e.g., viewing a map with or without Points of Interest annotations.
- Like any other view computation function inputs, perspectives are free to affect the result of the function in any way. This includes the portion of the user interface that projects a view of model state. For example, perspectives can be used to highlight a portion of a model's projection to identify a user's selection. This is done without the application's knowledge. If an operation needs to be performed on the selection, the controller can identify the selection and pass it to the application as an operation parameter.
- A data perspective could be used to actually *warp* the view of a model, in order to represent tentative changes to the model that do not actually affect the model until they are committed. The original model data does not change, but the user's view of it is modified as specified by the perspective. This could be used, for example, to type proposed text directly into a document, or to make proposed changes to a drawing or 3D model. The permanency of the changes would be subject to the application's acceptance at commit time.
- Data perspectives can be shared read-only with other participants' views in order to compute group awareness widgets⁴.

⁴Group awareness widgets are user interface components that give participants information about

- Data perspectives can be used to specify positioning and iconification of windows in a workspace.

3.4.4 Timer Perspective

In certain cases it is impossible to send model change notifications to views with the necessary frequency and precision, because of network latency, bandwidth, and unpredictability in transmission times. For example, sending frame change events over the network would not work well if participants were watching a movie or animation together. In these cases, we can take advantage of the fact that all computers have replicated clocks whose increments (and even values) are highly synchronized. Special distributed and automatically incrementing perspectives can be set to trigger events at regular, fine-grained, and precise intervals, thus synchronizing all participants' views. The application can turn timers off and on to implement pause and play operations for a movie. Irregular intervals for fine-grained simulations can be represented similarly. In this case, the application can stream irregular timer intervals to timer perspectives used for simulations so that these intervals can be triggered precisely on time. These timer perspectives need access to a clock input in their container. The primary difference between data perspectives and timer perspectives is that the timer perspective has object-specific behavior, which is triggered automatically.

3.4.5 Mobile Model

A mobile model is similar to a data perspective, except that it is visible to, and can therefore influence, the application. Since the application has access to a mobile perspective, it can also persist its data. Some application data need not be hidden

what other participants are doing.

behind application-specific operations. For example, a paragraph or sentence in a document or a list of node positions in a graph can be easily manipulated with a generic data structure using a standard data representation facility. The advantage of doing so is that the data can then be physically separated from the application (which may have more restricted mobility) and migrated to containers that need know nothing in particular about the data and how it is being used. Migration (for all mobile entity types) will be further discussed later in this chapter.

3.4.6 Immutable Model

An immutable model represents static (constant) data. Static data is easy to reproduce because it can be maintained at the central location and copied to any distributed location as necessary. It therefore doesn't need to be captured from a distributed site and replicated elsewhere when, e.g., a user moves. Immutable model entities are visible to the application, and may be quite large. Caching subsets of large models is discussed below. An example of an immutable model is map data for a mapping application.

3.4.7 Immobile Model

The immobile model entity class is a catch-all for all models that cannot be easily migrated due to excessive container requirements, location-dependent external references, size, the need to maintain execution state, the need for application-specific operations, or other reasons. Immobile models are represented centrally and are modified only by applications, in response to either external inputs or operations applied to an aggregated controller.

3.5 Data Caching

Both mobile and immobile entities can be replicated in a master/slave relationship (Figure 3.11). Master replicas can be modified directly by an application or

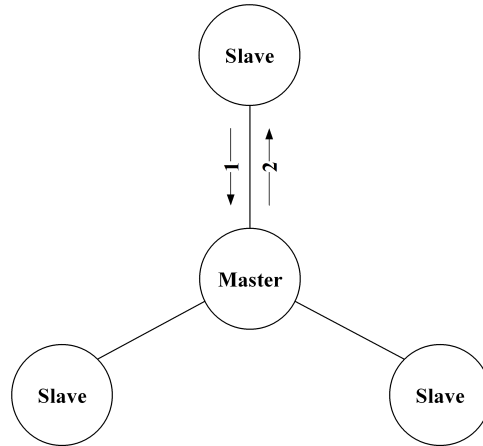


Figure 3.11: Master/Slave Replication in a Centralized System

controller. Slave replicas are only updated by master replicas, and they are kept as up to date as possible at all times. However, slave replicas need not contain all the data that their master replicas contain. They can act as data caches, filled on demand according to view computation function needs. For example if two participants are working independently on a document, only those portions of the document visible to a particular participant need be represented in his slave replica.

3.6 Mobile Entity Migration

Data perspective migration enables perspectives to quickly migrate from floor holder to floor holder, where the “floor” can be switched explicitly, or can be implicitly determined by the infrastructure based on participant activity. Migration can happen extremely quickly, because perspectives can be replicated in a master/slave

relationship as discussed earlier and shown in Figure 3.11. (Note that latency is twice the network transmission time for all participants in this diagram.) The result of such a migration is shown in Figure 3.12. No actual data must be switched when the floor

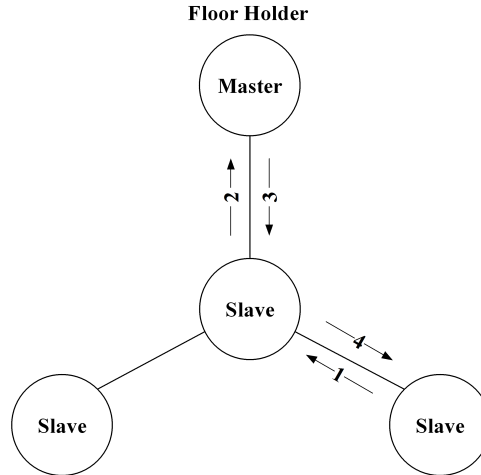


Figure 3.12: Master/Slave Migration in a Centralized System

changes hands. (Note that latency is zero for the current floor holder, and four times network transmission time for all others.) Data perspectives are migrated independently, so that different perspective master replicas can be near different participants at any given point in time.

Timer perspectives need not migrate; their entire purpose is to provide state change for a particular participant, in synchrony with other participants.

Mobile models can migrate similarly to data perspectives. The main difference is that the model application observes the central slave replica of the mobile model so that it can react to its state changes (Figure 3.13). Applications can also explicitly demand the master role for the mobile model for a time, should they need to gain more control over the model for synchronization or other reasons. For example, an application may wish to become the master of all the paragraph mobile models in a document so that it can perform an atomic global replace operation in the entire

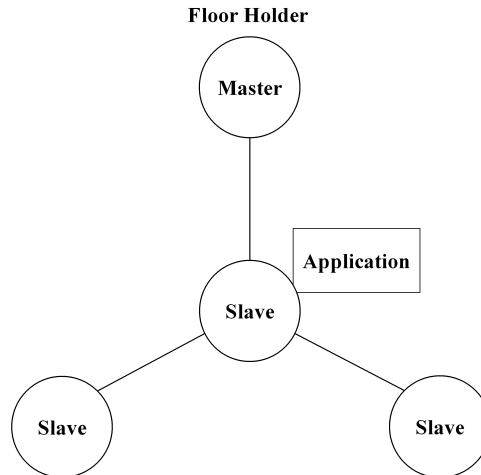


Figure 3.13: Mobile Models

document. The mobility of mobile models solves the most egregious latency problems in centralized systems, where such operations as typing or moving objects using direct manipulation are too slow because of frequent round trips to the central model.

Let's revisit Figure 3.12 for a moment. If the master is not in the center and its machine or connection dies, the central slave will have a reasonably up-to-date version of the perspective or model, and can take over or reassign the master role. For data perspectives, having the most up-to-date value would normally be unimportant, since these interactions are transient and are never seen by the application. The lost up-to-date value at the failed master would most likely represent interactions of the floor holder, in which case the floor holder may have invoked the lost operation(s) and may have even seen feedback from those operations (which other participants would not have seen). In the latter case, the feedback seen would be rolled back when the floor holder participant rejoins the conference. If the lost updates were from a non-floor-holder they would represent actions taken by the non-floor-holder participant for which he has not yet seen feedback (but for which the floor holder may have seen

feed-through⁵). In this case and in the former case where the floor holder has not yet seen feedback from his interaction, the lost interactions would act as no-ops from the participant’s perspective.

For data perspectives, if this degree of fault tolerance is not necessary, the central slave can be eliminated and the central component can simply route perspective operations. In this case, perspective state may simply be lost, e.g., if there are no slaves and the master dies. The perspective can then be reset to a default state. This might be appropriate, e.g., for scroll bars or selections. One advantage of this special but common case is that if a perspective is not being shared at all, only a peripheral master copy need be maintained, eliminating all network traffic and central node processing for that perspective.

In the end, the centralized architecture has been adapted so that there are many independent centralized architectures, one for each data perspective or mobile model. This dramatically reduces latency issues and also spreads some of the load around, reducing classical centralized system bottlenecks. Immobile models would need to remain in a static central location, but mobile entities would be free to change the “center” (master) location freely and quickly. There would still be a static physical “center” through which all non-local communication would pass, but the current master would be the authority on the value of the entity. Usually social protocols and fine-grained entities will eliminate the cases where more than one participant is interacting with the same mobile entity simultaneously, but when this does happen the infrastructure can migrate the entity to the physical center to attain fair, consistent responsiveness for all users.

Migrating entities are often a better solution than locking, because locking implies

⁵Feed-through is evidence one participant sees of another participant’s actions.

that everyone without the lock must wait. With migrating entities no one need wait for a lock, though responsiveness for non-floor holders will suffer. Migrating entities are also much easier to implement and are more predictable for users than replicated schemes, because concurrency control is accomplished through serialization at the master.

3.7 Entity Classification Summary

In essence, one proposed strategy for making classes of entities easily and quickly migratable is to separate code from data (the opposite of what the object-oriented paradigm promotes), which reduces or eliminates requirements to copy execution state and internal state and reduces container requirements. Data is represented using a single standardized data representation facility, eliminating the need to migrate object class definitions to distributed containers. Other strategies are to limit entity size, reduce or eliminate entity dependence on external references and application-specific operations, reduce or eliminate the application's dependence on entities, and take advantage of immutability.

Making view computation and controller migration easier and faster results in more robust and snappier latecomer and mobility support. Doing the same for data perspectives and mobile models further enhances the robustness and speed of latecomer and user mobility operations, and additionally improves overall interactive responsiveness.

In centralized synchronous distributed collaborative systems, I have only found two examples of support for migration to increase responsiveness for floor holders. One is XTV/Chung's infrastructure, which does process-level migration. The other is GEN[O'G98], which demonstrates implementing the migration of object-oriented

objects as an example of that toolkit’s extensibility, but does not further explore migration. Both are heavier-weight solutions than migrating perspectives and models.

In addition to simplifying and speeding migration, entity classification provides a framework for exposing fine-grained divergence possibilities to developers and users. View computation functions and controllers can be exposed to users so that they can select wholesale divergence of views or operation bindings. Data and timer perspectives represent finer-grained UI state divergence opportunities. Mobile and immobile models are less apt to be good divergence units, though they may be in some cases. View computations, controllers, perspectives, and models can have schemas (types) associated with them, which can enable the infrastructure to limit the range of user choices to ones that are likely to make sense.

Chapter 4

Concur Requirements and Architecture

4.1 Requirements

Based on the discussions of the previous chapters, the requirements of my proof of concept software, Concur, are as follows:

- Must support individual work as well as all four classes of collaborative work illustrated in Figure 1.1.
- Must support transparent transitions among these modes of work.
- Must support latecomers and mobility.
- Must be deterministic.
- Must achieve application software and user mental model simplicity comparable to centralized systems.
- Must achieve latencies comparable to replicated systems.
- Must support a wider range of understandable, user-selectable divergence scenarios than previous collaborative systems.

The following section lists and elaborates upon the architectural choices I made in order to meet these requirements.

4.2 Elements of a Solution

The primary elements of my solution are:

1. The *push* Model-View-Controller paradigm,
2. A logically centralized architecture,
3. A common hierarchical data modeling facility,
4. Continuously evaluated functional views,
5. Perspectives,
6. Declarative user interfaces, and
7. Composition functions.

Each of these elements is discussed briefly in the following subsections.

4.2.1 The *Push* Model-View-Controller Paradigm

The classic Model-View-Controller (MVC) paradigm[GR83] (Figure 4.1) is a software engineering technique designed to promote software reuse. It does so by separating a software system into model, view, and controller components representing an object's state and behavior, a computation of a view (presentation) of the object, and a means of mapping user inputs to operations on the object, respectively. Thus, a model can be reused with different views, a view with different models, etc.

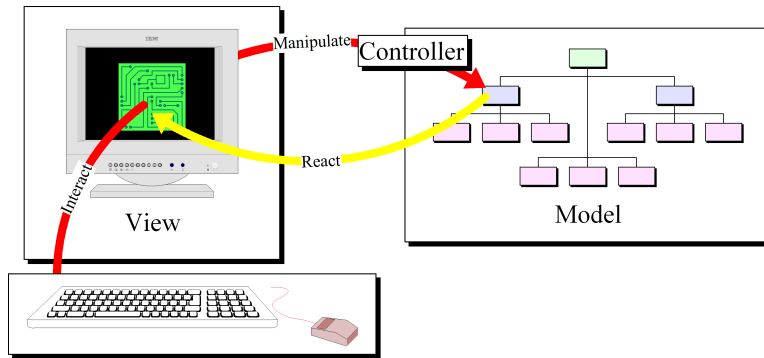


Figure 4.1: The Model-View-Controller Paradigm

In collaborative systems there are additional motivations for using this technique, because it supports multiple and potentially different simultaneous views of the same model (Figure 4.2), reduced or eliminated collaboration awareness in the model, and distributed systems where the model and views are on different machines.

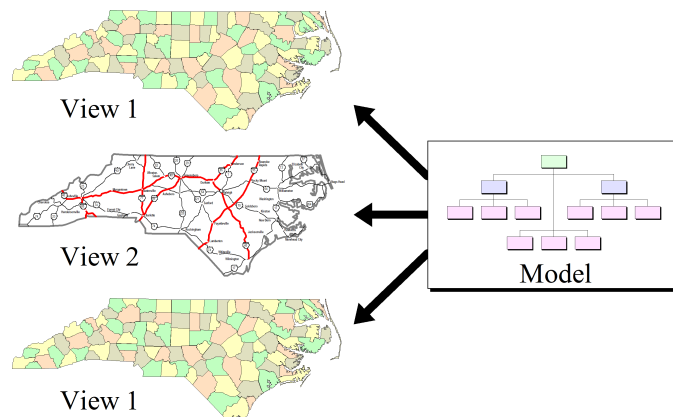


Figure 4.2: Multiple Views of a Model

The original MVC variant works like this:

1. A model is instantiated with no views attached.
2. Views are instantiated, which register themselves with the model. For each view, a controller is instantiated, which attaches itself to both view and model.

3. A user interacts with the view presentation. The user's interactions are mapped to model operations by the controller.
4. The model notifies all registered views that something has changed in the model.
5. Each view queries the model to determine what has changed, and updates itself accordingly.

The interaction described above is called the *pull* paradigm, because the view “pulls” information from the model to determine what has changed. This paradigm minimizes dependencies between the model and the view, because the model need not know anything about a view other than the fact that it is interested in the model. The disadvantage of the *pull* variant is that it can be quite inefficient for the view to determine what has changed in the model.

For efficiency, and to simplify views, a *push* variation on this theme is sometimes used¹. In this case the view specifies an interest in *particular* changes in the model, and the model, in turn, pushes descriptions of the changes that actually occur (i.e., *change notifications*) to the views interested in those particular changes.

The *push* MVC variant works like this:

1. A model is instantiated with no views attached.
2. Views are instantiated, which register themselves with the model, expressing interest in particular types of changes in the model. For each view, a controller is instantiated, which attaches itself to both view and model.
3. A user interacts with the view presentation. The user's interactions are mapped to model operations by the controller.

¹Both the push and pull variants are discussed in [GHJV95a].

4. The model determines which views are interested in the particular changes that have occurred, and sends a notification to each of these views, describing the changes in detail.
5. Each view updates itself accordingly.

This *push* variant can be a great deal more efficient than the original *push* variant, because views are not notified of changes in which they have no interest, and, more importantly, because views are told exactly what has changed so that they need not deduce this information. The cost of this approach is that the model must be more complex and must have more knowledge about views, since changes in the model somehow need to be matched with the interests of the various views. However, this complexity and view knowledge can sometimes be encapsulated in a reusable data modeling facility, as described in Section 4.2.3.

The Observer Design Pattern[GHJV95a], discussed in Section 1.5, is a generalization of the MVC paradigm, where views (observers) can themselves be models (subjects), and this pattern can be chained into multiple software layers (Figure 1.15).

4.2.2 Logically Centralized Architecture

Several of the requirements listed in Section 4.1 motivate a logically centralized architecture. Of these, the most important are determinism and simplicity of the user mental model.

Replicas implemented as state machines are difficult to keep in sync with each other, even if they are not allowed to diverge for performance reasons. Tight control must be maintained over the inputs of the various replicas, so that they do not diverge due to differing inputs. The slightest difference in inputs can cause different replicas to land in different states. It is usually difficult to determine that this divergence has

happened, and divergence may not become apparent until an arbitrarily long period of time has elapsed. For the same reason, support for latecomers and mobility are much easier in a centralized architecture. A centralized master model can be used to persist application state while an individual moves from one location to another, and it can be used to reconstruct a mobile user's or instantiate a latecomer's presentation.

Thus, centralized architectures simplify the implementations of both applications and infrastructures. While complexity trade-offs can be made between the two (e.g., hiding synchronization issues from applications by implementing all synchronization in the infrastructure), it is not always desirable for applications to be synchronization unaware. In any case, the sum of the complexity in infrastructure and application is greater for replicated systems than for centralized ones.

When replicas are allowed to diverge for performance reasons (which is the primary motivation for replicated collaborative architectures), accurately maintaining an understandable notion of synchronization becomes much more difficult. Replicated systems are capable of providing fast, local interaction with users because it is possible for the user to interact with the local model with no dependency on network latencies. However, this speedup comes at a cost with respect to the user's mental model. In general, it is impossible to maintain divergent replicas without negatively impacting the user experience. If two model replicas are presented to the user as representing the same object, Aristotle's Law of Identity ($A \text{ is } A$) [Ari08] would require that they be indistinguishable. The fact that they are not indistinguishable indicates that, rather than being the same object, they are, in fact, different objects that are similar in some sense. (Similarity might mean, for example, that differences can be easily identified and automatically reconciled.) This creates confusion for the user, who is encouraged to view two objects as being the same, while his senses tell him otherwise. Furthermore, automatic reconciliation of divergent objects can cause

unexpected state changes, from the users' perspectives (e.g., the reconciliation might result in a state that neither user intended). For these reasons, our desire for a simple user mental model helps to motivate a centralized architecture, where there is a clearer notion of object identity. Centralized architectures are much easier to control in this regard, because they continually maintain a unique master copy of the model state.

Note, however, that a simple user mental model and a clear notion of object identity do not necessarily imply WYSIWIS collaboration. People are accustomed to mental abstractions of the physical world in which they live. It is perfectly reasonable for the object being shared to be some underlying abstraction instead of a particular projection of that object onto a display. For example, it is reasonable to share a notion of the current time between two users, who variously view the time via analog and digital clocks, or to share a document abstraction where each user is scrolled to a different position in the same document. The users understand that there is an underlying, shared time or document, even though they do not see it the same way (time) or do not see all of it at once (document).

4.2.3 Common Hierarchical Data Modeling Facility

In order for an infrastructure to provide collaborative capabilities to an application, it must have access to either the state of the application's model and/or view components, or to the outputs emitted by these components. For reasons that will become clear in Section 4.2.4, Concur takes the state approach.

There are two main approaches to giving the infrastructure access to the application state. Either the infrastructure can be provided direct access to the application state as it is naturally represented by the application, or the application can represent

(some of) its state in an infrastructure-determined form. The first approach requires less work on the part of the application developer, while the second makes the infrastructure less dependent on particular application technologies (e.g., programming languages and environments).

Concur takes the latter approach, for a number of reasons:

- **Technology independence.** Standard data representation languages tend to outlive programming languages and runtime infrastructures. A standard data representation language helps to isolate the collaborative infrastructure from these technology changes.²
- **Change notifications.** Using a common data representation enables Concur to ensure that change notifications are supported by the data structures. This relieves the application from having to code view registrations and change notification events. The common data representation thus encapsulates the model/view protocol for the application, so that the model need know nothing about views. This is especially valuable when using the push MVC design pattern.
- **Published vs. unpublished data.** Requiring the application to separately represent model data that might contribute to a view (published data) supports an important data hiding capability at the component boundary, similar to public/private distinctions at the object boundary. This avoids both having to make private data public so that it can be accessed by the infrastructure, and

²The Concur proof of concept actually does not use a *standard* data representation language. The Document Object Model[Mar02] (DOM) was originally used, but it was found to have certain deficiencies with respect to the representation of hierarchical data structures. For example, reparenting operations are mapped to remove and insert operations, which are awkward and inefficient for an observer/view to re-map back to reparenting operations. In the interest of expediency, I chose a hierarchal data representation API that was convenient for the chosen programming environment.

using separate mechanisms to avoid exposing public data that should not be exposed outside the component (unpublished data).

- **Distribution.** Concur distributes model state and view code, which generates view state based on model state. View code must run in a restricted container. If the model and view state were represented using the application’s native representation, infrastructure for maintaining that representation would have to be distributed as well, increasing container requirements.
- **Testing, Debugging, and Scripting.** A common data modeling language supports testing, debugging, and scripting in a manner that is independent of application technologies and less dependent on particular applications.

A common data representation language must support arbitrary application data structures. This implies that the language must support arbitrary graphs. On the other hand, hierarchical data structures are extremely common in applications, and it is sometimes helpful to the infrastructure if it can assume a hierarchical view of the data. For example, if the infrastructure needs to identify a portion of an arbitrary graph, it can only do so by maintaining a set of nodes and links defining that portion, or a computational rule defining that set. On the other hand, if it can assume the graph is a hierarchy (or directed acyclic graph (DAG)), it can often identify a sub-portion of the graph by simply specifying one (root) node. (In Figure 4.3, *Chapter 1* can be used to refer to all the shaded nodes.) Like file systems, Concur uses a representation language that assumes a basic hierarchical structure with a special link type for specifying non-hierarchical graph structures (Figure 4.4) as an overlay on the hierarchy.

In Concur, a common hierarchical data representation is used to represent both model (view function inputs) and UI state (view function outputs). Low-level user

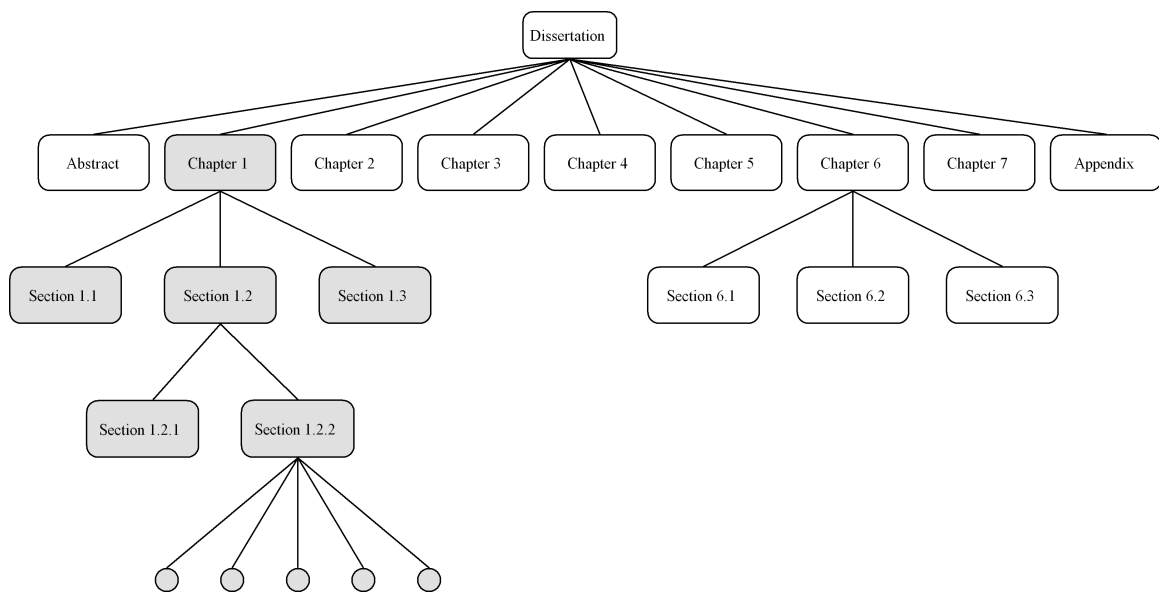


Figure 4.3: Hierarchical Data Structure

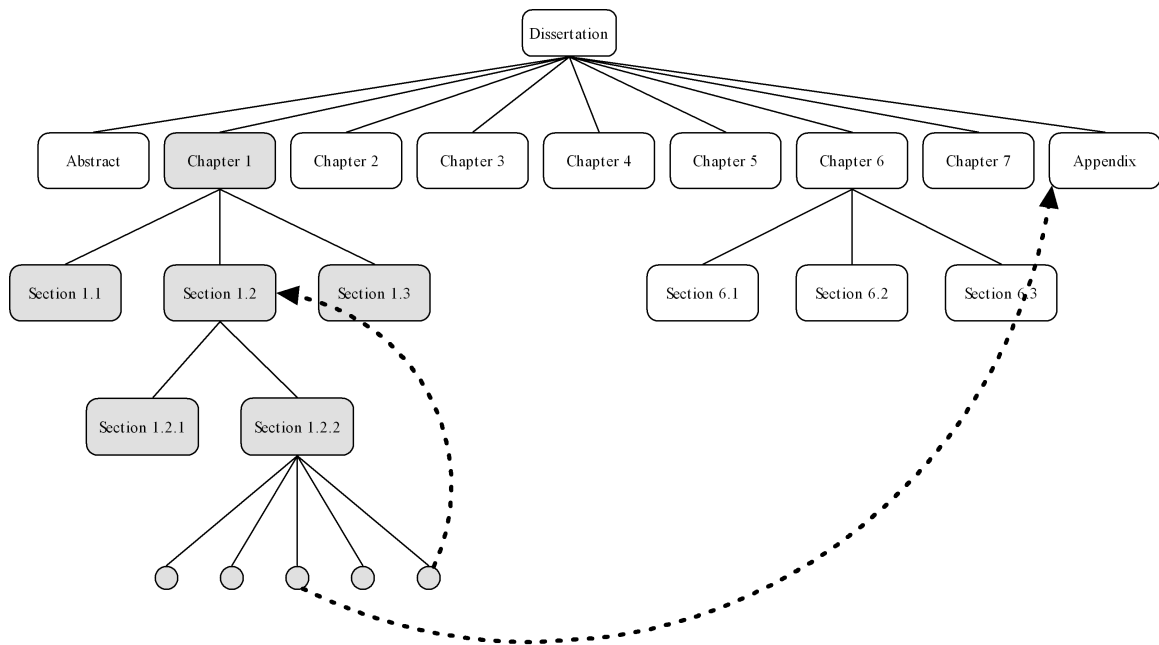


Figure 4.4: Graph implemented as Hierarchy with Special Links

events such as keyboard and mouse events must be able to be applied to the declarative UI data structure representing the UI. Concur implements an event mechanism similar to those used by the X Window System[SGR92] and the Document Object Model (DOM)[Mar02]. Events can be applied to any node in the hierarchy, and interest in these events can be registered (by a controller) with respect to any node. Events bubble from the node to which they are applied up to the root, notifying observers along the way (Figure 4.5). The same mechanism is also used for registering and notifying view functions of changes in its model or perspective inputs. Concur

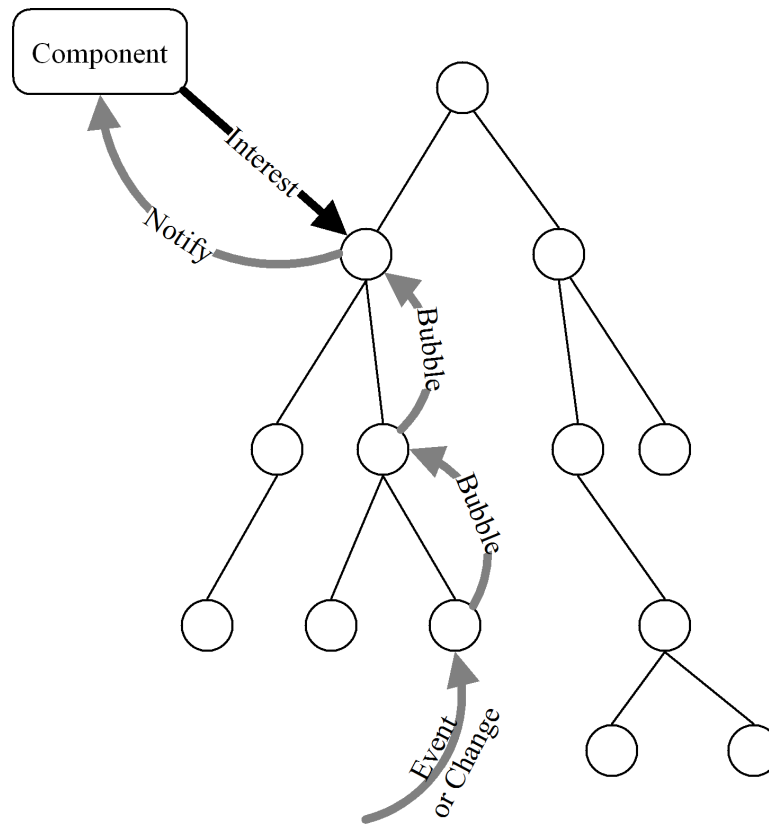


Figure 4.5: Bubbling of Events or Changes

also requires that the set of (DAG) inputs to a continuously evaluated function always remain in the domain of that function. Concur's data representation therefore imple-

ments transactional modifications to the data structure, enabling multiple changes to be made before event notifications are released.

4.2.4 Continuously Evaluated Functional Views

As described in Section 2.3.2, models and views are often best viewed as state machines. The state of these components is established by either directly setting the state (via transfer of all or part of the state from a replica) or by the execution of commands (state machine inputs) which modify the component state in some deterministic fashion. Initial state is often established via state transfer, after which the state is maintained via command execution. Assuming an initial component state S_0 and subsequent states S_i that result from the execution of commands C_i , the state of a component after a command execution can be viewed as a function f of its previous state and the command to be executed:

$$S_{i+1} = f(S_i, C_{i+1}) \tag{4.1}$$

By induction, another form of this equation is:

$$S_{i+1} = f(S_0, C_j | j \in \{1..(i+1)\}) \tag{4.2}$$

That is, the current state of the component at any point in time depends on its initial state and the ordered set of commands that have been executed.

Non-determinism comes about when we do not have control over all of the inputs to the state machine. That is, the component actually operates deterministically, but it does not appear to do so from our perspective because we do not have access to all

of its inputs. If we represent all such uncontrolled inputs as $\mathcal{X}$, Equation 4.2 becomes:

$$S_{i+1} = f(S_0, C_j | j \in \{1..(i+1)\}, \mathcal{X}) \quad (4.3)$$

In order for us to make guarantees about the behavior of our system, we must either assume that components to be replicated are coded such that they have no dependencies on external inputs (Equation 4.2), or that their behavior is well-enough understood that a careful choice of $C_j | j \in \{1..(i+1)\}$ will avoid any non-determinism introduced by $\mathcal{X}$.

For completeness, we need to consider not only the state S of the component, but the outputs O emitted by that component on state transitions. If we simplify the equation by assuming that outputs are only emitted on the execution of commands C (which we can do without affecting the results of this discussion), the output equation corresponding to Equation 4.3 is:

$$O_{i+1} = g(S_0, C_j | j \in \{1..(i+1)\}, \mathcal{X}) \quad (4.4)$$

Here again, we must assume we can eliminate the affect of any uncontrolled inputs. But since we must replicate the component multiple times, we must further constrain the outputs to be idempotent.

The salient point of the preceding discussion is that all collaborative infrastructures must make assumptions about the coding of components they replicate. Where these components are state machines, we must assume that there exists a sequence of commands that will deterministically put the replica into the desired state, independently of any uncontrolled inputs, while producing only idempotent outputs.

Concur takes a fundamentally different approach to the requirements imposed on

the view components it replicates. In Concur, a view is best modeled as a *continuously evaluated function of known inputs* rather than as a state machine. A continuously evaluated function is one whose output is always, modulo computation time, a function of its inputs. The functional requirement means that the same inputs always produce the same outputs. This property is known as *referential transparency*.

Thus, in Concur, there is no state S , there are no uncontrolled inputs $\mathcal{X}$, and commands C are replaced by inputs I . The relevant view equation for Concur is:

$$V_{current} = v(I_{current}) \quad (4.5)$$

That is, the current output V is a function v of the current inputs I . Uncontrolled inputs and non-idempotent outputs are also eliminated by the restricted container in which Concur view and controller code runs.

The requirement set imposed on the coding of functional view components is not substantially more burdensome than the requirements described above for components modeled as state machines. However, it has the following dramatic advantages:

- Latecomer support is simplified, because the infrastructure need never store the state of a view or the commands required to establish that state.
- Establishing a latecomer view can be more efficient, because it depends only on a bounded set of current inputs, not a sequence of commands that can increase in length indefinitely over the life of a session.
- Mobility support is simplified because there is no view state to maintain while no one is viewing an object.
- The robustness of the system is enhanced because the current view output does not depend in any way on previous state or inputs. Thus, if there is a

view code bug that causes the output to be incorrect for a given input, future outputs that are correctly computed are not affected. By contrast, in the state machine approach a view code bug can create incorrect internal state at one point in time that can invalidate future states and outputs indefinitely, even if the computation of these latter states and outputs is correct.

- Testing of the replica’s adherence to the infrastructure’s requirements is facilitated, because setting up the current inputs to a function is simpler than establishing the state of a state machine, and because functional guarantees can be checked without knowing the semantics of the function itself.

4.2.5 Declarative User Interfaces

The output V of the view function specifies *what* the user interface should look like at any given point in time (not *how* it should be constructed). Specifying the UI declaratively simplifies the coding of the view function v by off-loading the *how* of its construction to the Concur infrastructure. It also provides an efficient means of taking snapshots of the user interface, since the specification can be stored at a high level, rather than as a pixel-level image.

In order to guarantee that the functional characteristics of view computation are retained while mapping the output V to the display, the rendering of V is performed by a continuously evaluated *projection* function p , which maps V to a (normally contiguous) set of pixels P on the display:

$$P_{current} = p(v(I_{current})) \tag{4.6}$$

The function p is provided by the Concur infrastructure. The projection P can be

realized using any UI technology or any combination of UI technologies. Examples are HTML, XHTML[MK02], Windows GDI[WG02], X[SGR92], Java Swing[RV03], Tk[WH03], and OpenGL[SWND03].

In addition to rendering its model data in some fashion, a view must accept inputs from users and pass them to the application (model code). In Concur, the UI generated by the p function dispatches input events to the document V . Application-specific controller code, distributed by the infrastructure and executed in the same restricted environment that executes the view code, maps these UI-domain inputs to model-domain operations and passes them to the model code via a controller. Concur supports collaborative capabilities such as floor control in the infrastructure itself, optionally relieving the model code of this aspect of collaboration awareness. In this case, the infrastructure will determine which model-domain events to forward to the application’s controller instance from the various users’ controller instances. Alternatively, Concur can aggregate the model-domain operations from all users into the application’s controller instance, augmenting them with information identifying the users, so that the model code can implement such collaboration-aware features itself. In more elaborate applications, this is precisely what is desired. This strategy supports a full range of application styles, including various floor control strategies, programmatic access controls, and multiple simultaneous inputs from various users, while enabling applications to be completely collaboration unaware when that is desirable.

4.2.6 Perspectives

Concur supports multiple model inputs feeding into one view:

$$V = v(I_i | i \in \{1..n\}) \tag{4.7}$$

This is convenient for a number of reasons:

- It supports division of responsibility in model code by enabling different parts of the code to maintain different models.
- It supports code reuse by enabling different applications to present different data items, all of which contribute to the view.
- It enables data pertaining to the application domain to be separated from data pertaining to the user interface. For example, the former might be the text of a document, while the latter might be the positions of scroll bars.
- It enables views to diverge among participants, where some but not all of the model inputs are shared.

The inputs feeding a particular view are typed, because the view needs to be able to understand the structure of the set of inputs and of each input individually in order to understand the data embodied therein. Type matching is performed by the infrastructure at run time, to ensure that only valid view computations are constructed.

One of the contributions of this work is to propose a variation of the Model-View-Controller[KP88] (MVC) paradigm, which I will name Model-Perspective-View-Controller (MPVC). This adaptation of MVC identifies a sub-class of models called *perspectives*, which have the following characteristics³:

- Perspectives may have arbitrary structure, but they are typically small and simple.
- Perspectives are not accessible to the model code of an application. That is, perspective values cannot directly affect application logic.

³From this point forward, models and perspectives will be distinct in this paper.

- Perspectives are, however, accessible to views. Any influence a perspective may have on a model will come via the view and controller.
- Views specify which perspectives they need, but they do not allocate their perspectives. This task is performed by the infrastructure.
- Since perspectives are associated with views rather than models, they tend to have a shorter lifetime than models. They are not persisted beyond the lifetime of the view definitions referencing them.
- Perspectives may have certain basic collaborative services provided by the infrastructure, such as floor and access controls.
- Perspectives are shareable among multiple views, just as models are. However, if they are not being shared, they can optionally migrate to the host running the view code using them at the moment (i.e., the central master is not maintained, and the local replica becomes the master). If no view replicas referencing a perspective exist at some point in time (e.g. during a mobility transition), or if an unshared perspective becomes shared again, it migrates back to the central server. Perspective migration makes fast local interaction possible.
- Since perspectives are separate units of data that can be individually supplied to views as inputs, they represent units of coupling and divergence, where different users can either share or diverge on each perspective. The same is true for models, but they are less likely used as a unit of divergence independent of perspectives.
- Perspectives can be grouped so that they can be shared or diverge as a unit. This is useful, for example, for specifying whether a group of perspectives defining a user interface state are shared or not.

Taking this model/perspective distinction into account, Equation 4.7 becomes, for models M and perspectives π :

$$V = v(M_i | i \in \{1..n\}, \pi_j | j \in \{1..m\}) \quad (4.8)$$

Perspectives are useful for implementing a broad range of single-user and collaborative capabilities. Some of these are listed in Section 3.4.3. Many other possibilities exist for the use of perspectives. These are only listed as representative examples.

In summary, perspectives help to separate application domain code from the user interface, promote collaboration unawareness in applications, enable a wide range of divergence among views of the same data, facilitate group awareness, support local feedback that can be shared as needed, and support tentative manipulation of model data.

Given this background on perspectives, it will be instructive to list and discuss examples of the kinds of divergence supported by Concur, and how these are related to changing the view function v and the representative model and perspective inputs M and π , respectively. These examples are summarized in Table 4.1.

Consider the following equation:

$$V_0 = v_0(M_0, \pi_0) \quad (4.9)$$

This is the baseline equation, to which we will compare all possible combinations of differences in v , M , and π .

Now consider Equation 4.10:

$$V_0 = v_0(M_0, \pi_0) \quad (4.10)$$

Equation	Examples and Comments
$V = v(M, \pi)$	WYSIWIS. Model, perspective, and view function all shared.
$V' = v'(M, \pi)$	Digital and Analog Clocks. View functions (formats) diverge, but model (time) and perspective (time zone) do not.
$V' = v(M, \pi')$	Viewing the same geographical map independently. Model (map) and view function (format) are identical, perspectives (scroll bars and selections) diverge.
$V' = v(M', \pi)$	Scheduling an appointment with the same UI. View function (format) and perspectives (date & time selections) are shared, models (calendar entries) are not.
$V' = v'(M, \pi')$	Spreadsheet and graph of same data. Model (data) is shared, view functions (format) and perspectives (scroll bars) are not.
$V' = v'(M', \pi)$	Scheduling an appointment with different UIs (e.g., Outlook vs. Any-Time). Perspectives (date & time selections) are shared, view functions (UIs) and models (calendar entries) are not.
$V' = v(M', \pi')$	Code sharing of drawing editor with different drawings. View function (UI code) is shared, models (drawings) and perspectives (UI state) are not.
$V' = v'(M', \pi')$	No sharing. Completely different applications operating on different data with no perspectives to synchronize interaction.

Table 4.1: Summary of Divergence Possibilities

This equation is identical to equation 4.9, so it represents WYSIWIS collaboration (Figure 4.6). That is, if both users share the same view function v and inputs M and π , they will be sharing identical projections.

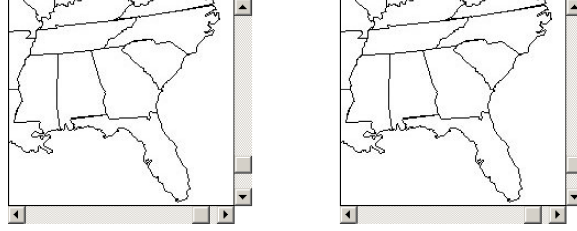


Figure 4.6: Same Model, Perspective, and View

Equation 4.11 represents sharing the same data (both model and perspective), where the view function computed on that data differs (Figure 4.7).

$$V_1 = v_1(M_0, \pi_0) \quad (4.11)$$

In this illustration the time of day represents the model, while the time zone (Chapel Hill) represents the perspective. Another example of this type of divergence would be viewing strip plots of the same data with a shared horizontal scroll bar, but with different views of the data (bar chart and line graph).

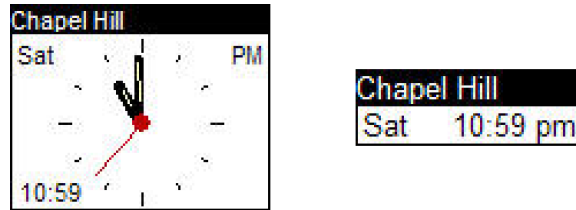


Figure 4.7: Same Model and Perspective, Different View

The next equation (4.12) diverges in the perspective, but not in the model or view.

$$V_1 = v_0(M_0, \pi_1) \quad (4.12)$$

This is illustrated in Figure 4.8, where the map (model) and the view function are shared, but the perspectives (scroll bars and selections) are not. This is a very common divergence scenario, other examples of which include viewing a 3D model from different viewpoints, maintaining separate UI state (e.g., menus), and viewing the same time via an analog clock from different time zones.

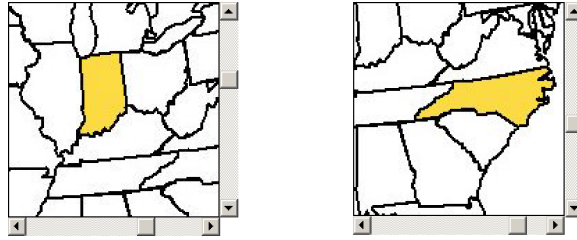


Figure 4.8: Same Model and View, Different Perspectives

Equation 4.13 represents an unusual scenario.

$$V_1 = v_0(M_1, \pi_0) \quad (4.13)$$

In this case, the model differs while the view and perspective remain the same. Figure 4.9 shows an example, where John and Kevin are looking at their own calendars (models) from the same perspective (date and selected time), in order to schedule a meeting together. Another example would be a quartet, where each of four musicians sees his own music (model) in a common format, and the point in time in the piece (perspective) is synchronized. A third example might be a card table, where the positions of the cards on the table are represented by perspectives, and the values of

the cards are represented by individual models for each user. In this way, cards a user should not be able to see can be represented by the model for the back of the card, so that the model data representing these cards is never sent to that user's computer.

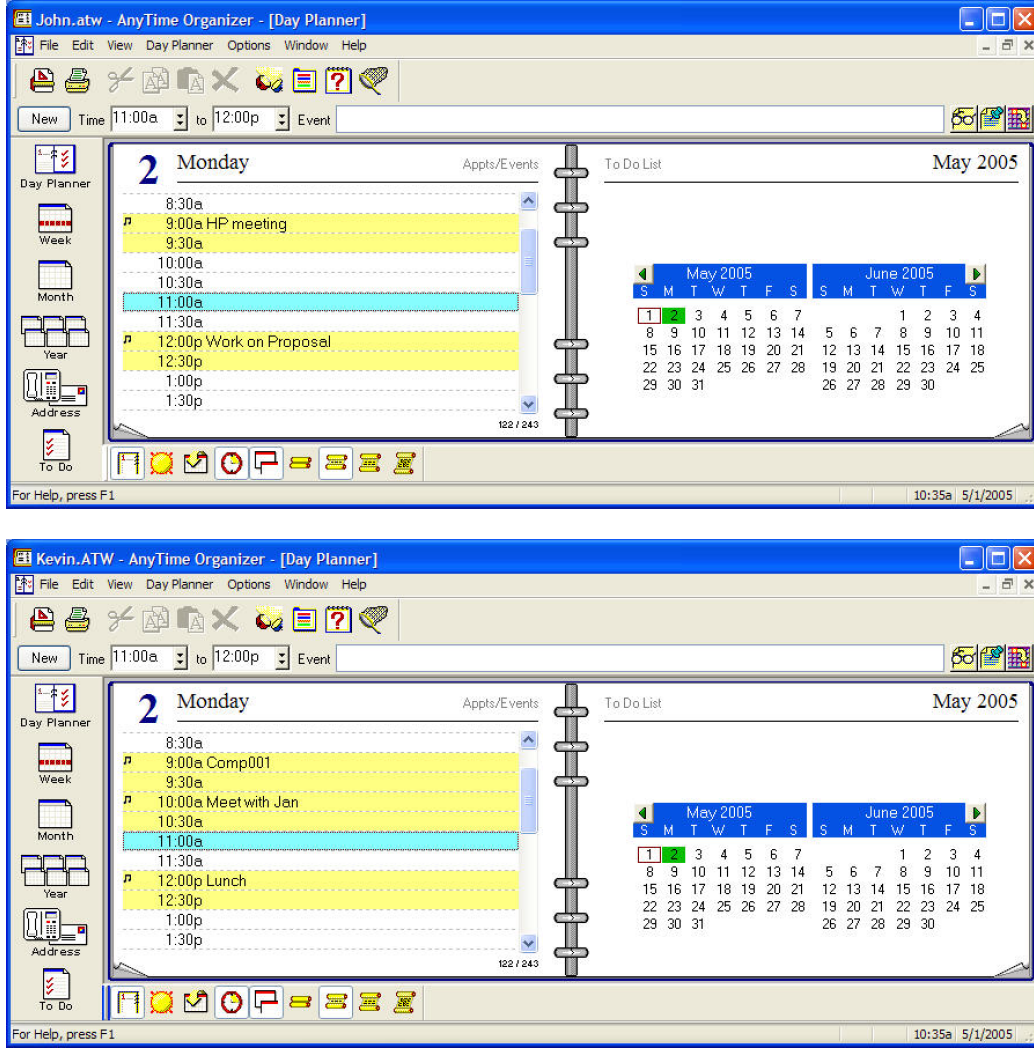


Figure 4.9: Same View and Perspective, Different Model

Equation 4.14 specifies that the model is shared, while the view and perspective differ.

$$V_1 = v_1(M_0, \pi_1) \quad (4.14)$$

This is a common way of viewing the same object in different formats, since perspectives often make sense only with respect to a particular format. Figure 4.10 demonstrates this scenario in terms of stock data and a corresponding stock chart. Another example is to view the same time (model) from different time zones (perspectives) via analog and digital clocks (views), respectively.

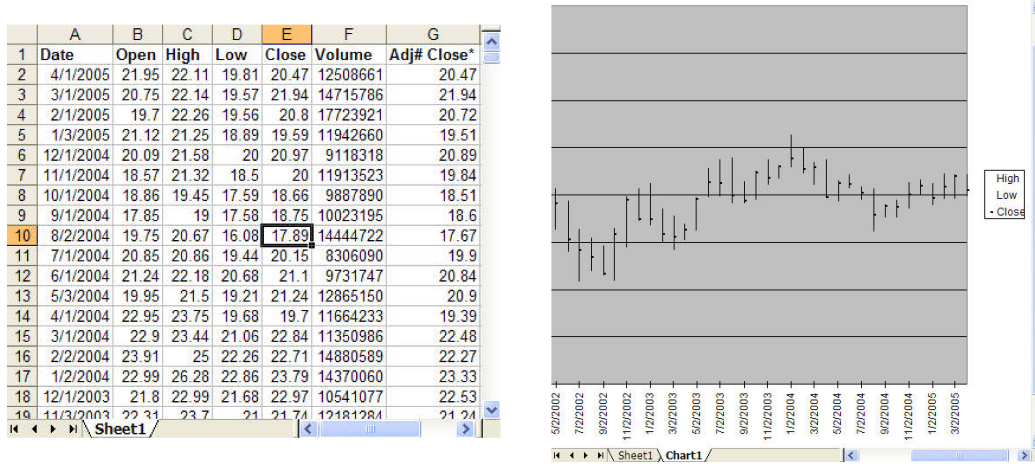


Figure 4.10: Same Model, Different Perspective and View

Like equation 4.13, equation 4.15 is unusual because the model diverges while other aspects are shared.

$$V_1 = v_1(M_1, \pi_0) \quad (4.15)$$

In this case, the perspective is shared, but not the view. To envision this scenario, examine Figure 4.11, which is a variant of Figure 4.9 where John and Kevin use different view software (Outlook and AnyTime) to view their own data, but share the date and time selection in order to schedule a meeting.

The next scenario, Equation 4.16, is much more common.

$$V_1 = v_0(M_1, \pi_1) \quad (4.16)$$

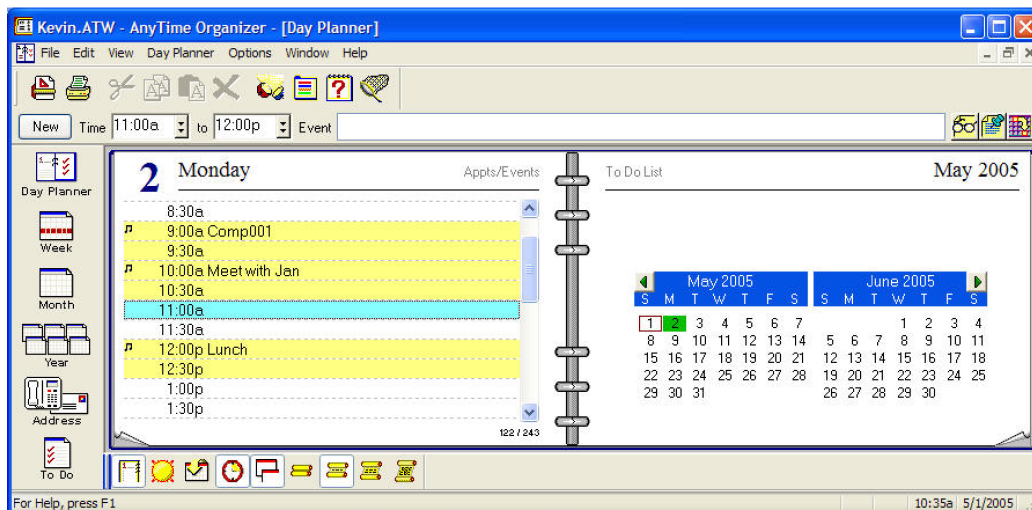
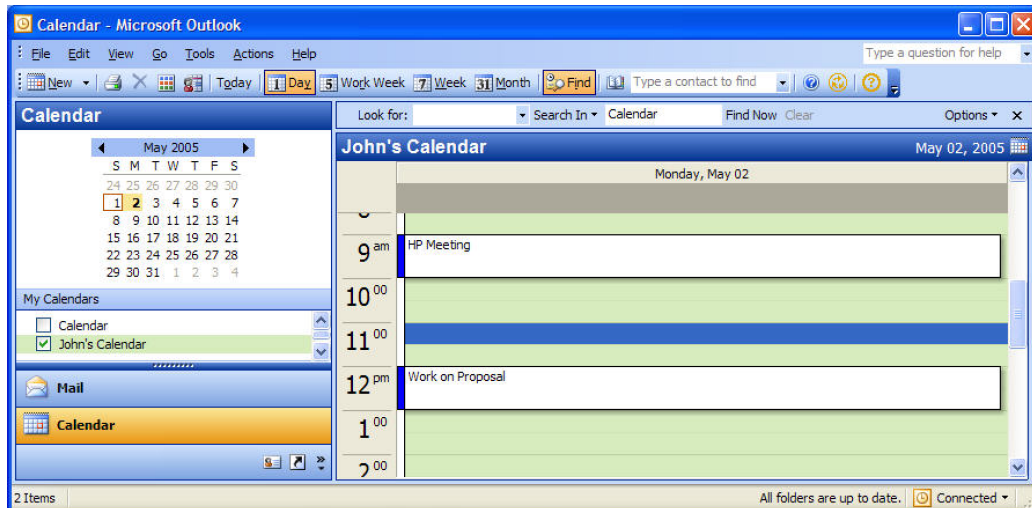


Figure 4.11: Same Perspective, Different Model and View

Here, the model and perspective are different, but the view function is the same. There is really no sharing in this scenario except for code sharing of the view. This is illustrated in Figure 4.12, where two users are using the same drawing editor to edit different drawings.

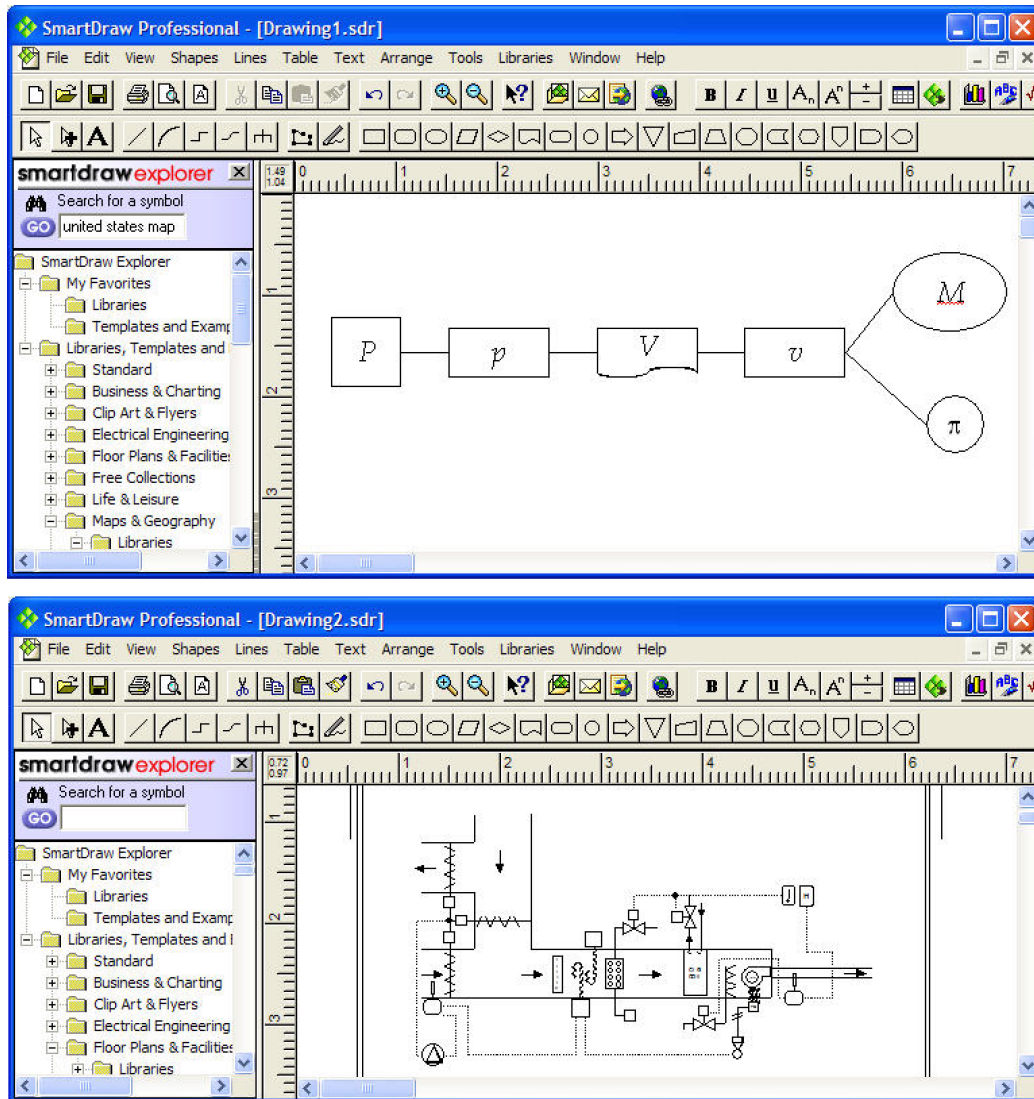


Figure 4.12: Same View, Different Model and Perspective

The final equation (4.17) is extremely common, but uninteresting.

$$V_1 = v_1(M_1, \pi_1) \quad (4.17)$$

In this case nothing, not even code, is shared. In Figure 4.13, the users are using different applications to view different data, and there are no shared perspectives for synchronization.

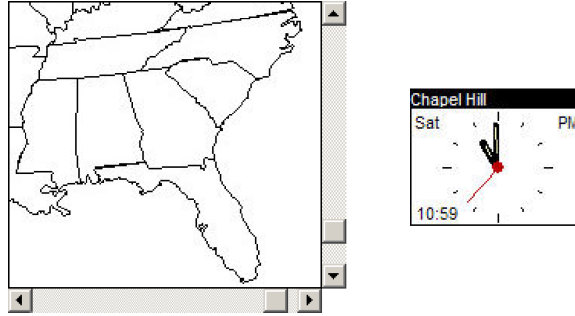


Figure 4.13: Different Model, Perspective, and View

This concludes our tour of divergence scenarios. Of course, there are infinite possibilities, since there can be multiple models M and perspectives π , which can be shared or not shared, independently.

4.2.7 Composition Functions

Views rarely exist in isolation. On a macro scale, the UIs of multiple applications are often grouped into a workspace. On a micro scale, individual applications are nearly always composed of multiple views, in order to simplify construction of the UI and facilitate code reuse. In Concur, view composition can be performed in two ways: a single view function can generate a hierarchy of UI elements, or view functions can be composed. The latter will be discussed in this section.

In the macro case, the enclosing view need know nothing about the enclosed view other than, perhaps, its size, which can be determined using standard negotiation techniques for widgets and window systems. That is, it need not know the identity of the view function $v_{enclosed}$, what its parameters $I_{enclosed}$ are, or even the (continuous) result of the function, $V_{enclosed}$. All it needs to do is to determine where it wants to put $V_{enclosed}$ within its own $V_{enclosing}$ declarative output structure. The function $v_{enclosing}$ need not copy $V_{enclosed}$ into $V_{enclosing}$, since $v_{enclosed}$ can be allocated its own portion of the $V_{enclosing}$ structure, where it can construct $V_{enclosed}$ directly. Revising equation 4.9 to include this style of view nesting, we have:

$$V = v(M_i|i \in \{1..n\}, \pi_j|j \in \{1..m\}, \mathcal{V}_k|k \in \{1..q\}) \quad (4.18)$$

where each $\mathcal{V}_k$ is a *view instantiation* that recursively takes the form of the right side of Equation 4.18.

In the micro case, embedded unknown views such as those described above for the macro case may also make sense under certain circumstances. Additionally, application UI views v may be second-order functions with view function parameters:

$$V = v(M_i|i \in \{1..n\}, \pi_j|j \in \{1..m\}, v_l|l \in \{1..r\}) \quad (4.19)$$

This gives the enclosing view more control over what is displayed in the enclosed view (because it can pass its own parameters to the enclosed view), and makes it possible to build applications that can import 3rd party view functions to display their data in different formats. The macro case can also make use of view function parameters, for example, to format window decorations. Thus, a window manager can be passed as a parameter, enabling divergence of window management.

Our final output equation, then, specifies that a view function v can depend on four types of parameters: models M , perspectives p , view instantiations $\mathcal{V}$, and view functions v :

$$V = v(M_i | i \in \{1..n\}, \pi_j | j \in \{1..m\}, \mathcal{V}_k | k \in \{1..q\}, v_l | l \in \{1..r\}) \quad (4.20)$$

Note that Equation 4.20 completely defines the current view output V with respect to the current value of all of its inputs.

4.3 Concur Architecture

All of the important concepts used in the Concur architecture have been covered in detail in the previous sections. In this section, I will pull these concepts together into a coherent discussion of Concur's architecture.

Figure 4.14 shows an overview of the Concur architecture. Figure 4.15 is the same as Figure 4.14, except that it has been tagged with numbered circles for reference in the following paragraphs. Note that we have also added a relational database to Figure 4.15, referenced from the model. The reason for this will be apparent in a moment.

The database (1) for our example contains weather data for Chapel Hill, North Carolina⁴. In particular, we will be looking at maximum daily temperature data, a small portion of which is shown in Table 4.2. The data in the database ranges from the year 1948 to the year 2001. Blank entries (e.g., for 1961-03-16) indicate missing data.

The model (2) is a software component whose purpose in this application is to

⁴This data is from the National Climactic Data Center[Nat08].

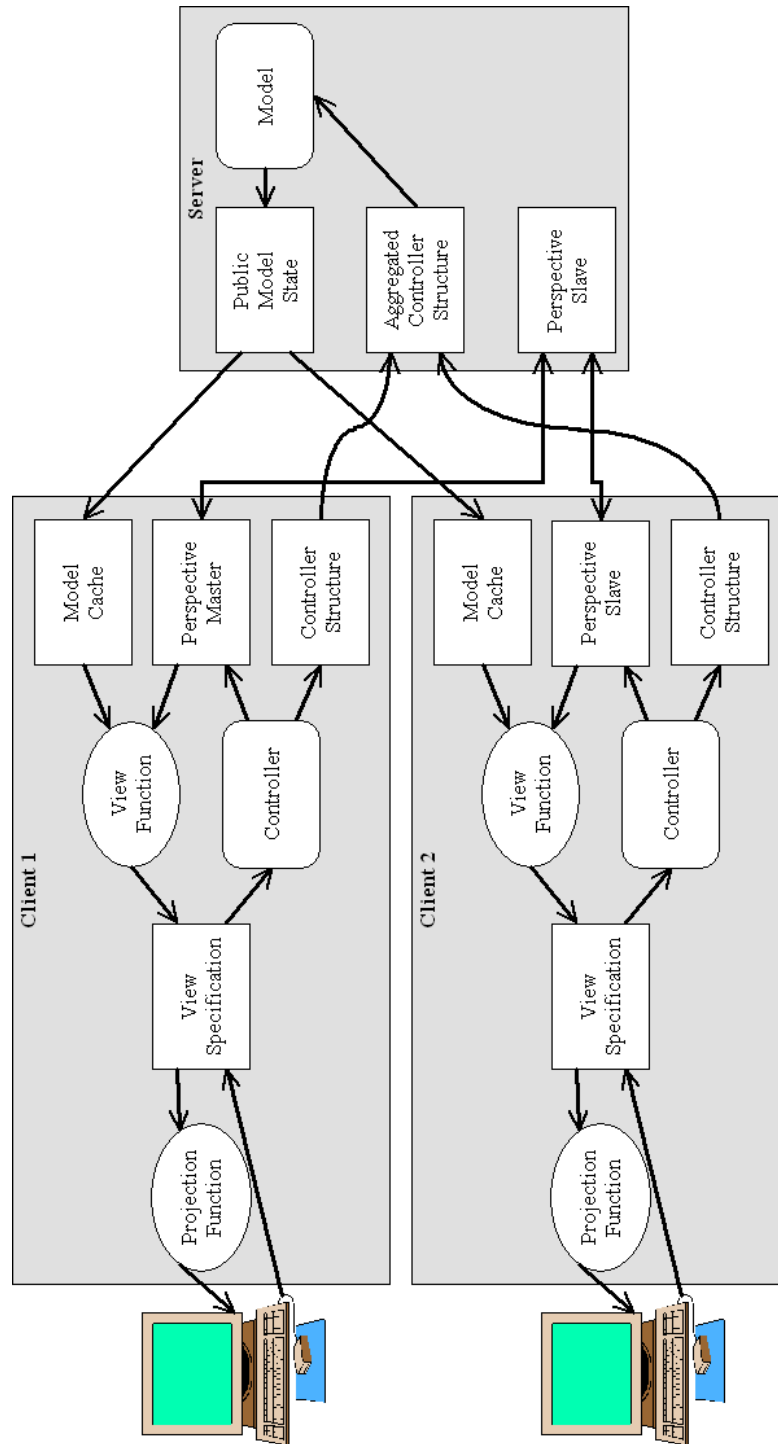


Figure 4.14: Concur Architecture

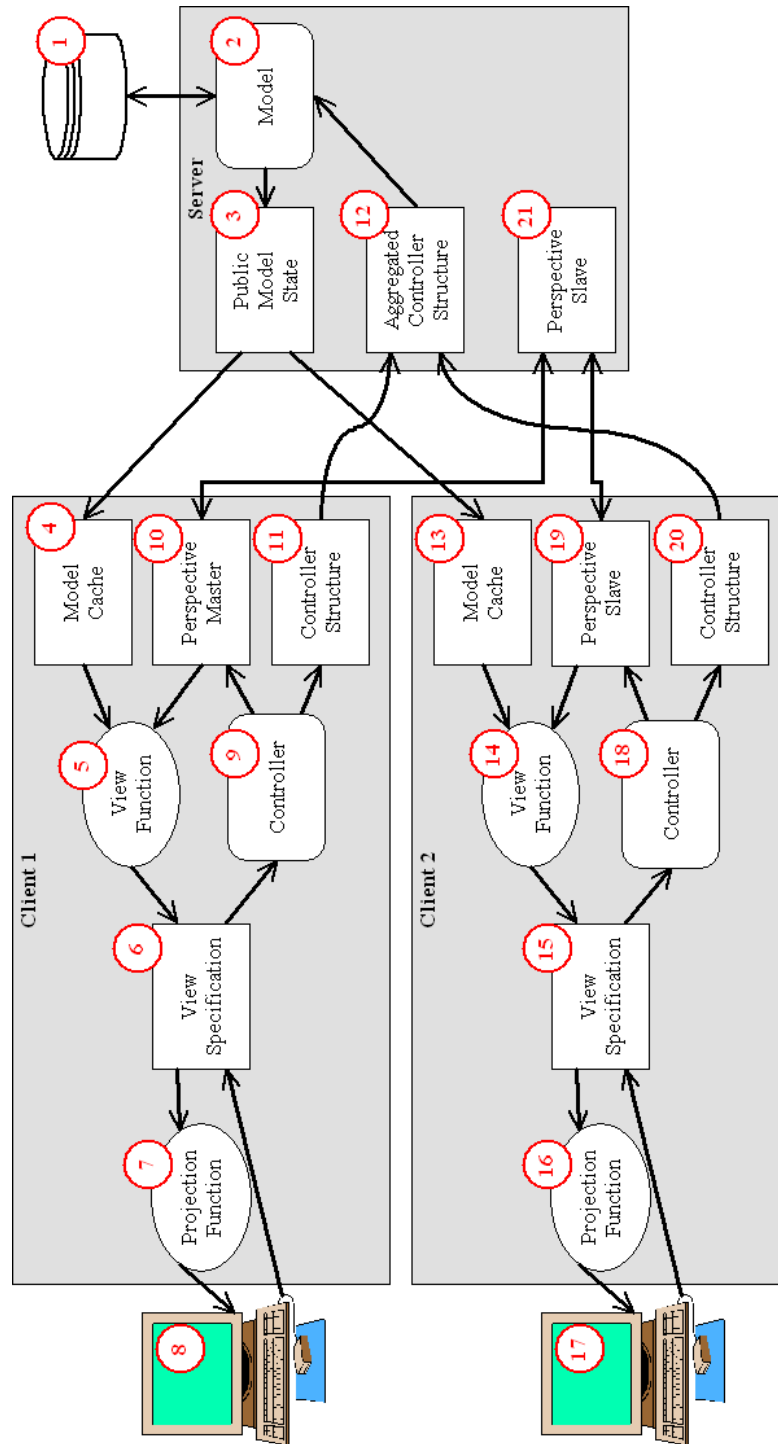


Figure 4.15: Concur Architecture with Tags

Table 4.2: Maximum Degrees Fahrenheit

Year	Month	Day	Maximum Degrees Fahrenheit
1961	3	1	69
1961	3	2	44
1961	3	3	57
1961	3	4	67
1961	3	5	79
1961	3	6	81
1961	3	7	79
1961	3	8	79
1961	3	9	72
1961	3	10	45
1961	3	11	51
1961	3	12	65
1961	3	13	74
1961	3	14	74
1961	3	15	71
1961	3	16	
1961	3	17	59
1961	3	18	47
1961	3	19	39
1961	3	20	66
1961	3	21	59
1961	3	22	42
1961	3	23	41
1961	3	24	59
1961	3	25	50
1961	3	26	63
1961	3	27	71
1961	3	28	76
1961	3	29	76
1961	3	30	76
1961	3	31	66

make the data in the relational database available for presentation using the Concur infrastructure. The relational database is an external reference for the model. If we assume the model is behind a firewall, this external reference restricts the model's mobility. There is therefore no disadvantage to implementing the model as a state machine.

The model makes a portion of the database public by representing its data in hierarchical form in the Public Model State data structure (3). This is an in-memory data structure built using Concur's common data representation API⁵. The Public Model State need not represent all the data in the database, only the data for which an interest has been expressed by an observer. However, let's assume that all the maximum daily temperature data in the database is represented in (3) for this example, since the data set is fairly small. Figure 4.16 shows a portion of the Public Model State.

At this point we have Public Model State but no observers of that state. Now let's assume Client 1 is instantiated. The View Function (5) is downloaded from the Server. The View Function then registers an interest in the Public Model State. For the moment, assume that the View Function is interested in the entire state; later we'll deal with the case where it is only interested in a portion of the state. All of the state is then downloaded to Client 1's Model Cache (4), which takes the form of Figure 4.16.

Next, the View Function is computed, generating the View Specification (6). The View Specification defines a projection of the model state, using the common data representation API. Figure 4.17 illustrates one form that this View Specification might

⁵In order to understand this example, we need a concise visual representation of the data structures built using this API. I will therefore map these data structures to XML[BPSM⁺04] for display in figures.

```

1 <WeatherData>
2   <State>North Carolina</State>
3   <City>Chapel Hill</City>
4   <AverageDailyTemperature Unit="Fahrenheit">
5     ...
6     <Year Year="1961">
7       ...
8       <Month Month="3">
9         <Day Day="1" Temperature="69"/>
10        <Day Day="2" Temperature="44"/>
11        <Day Day="3" Temperature="57"/>
12        <Day Day="4" Temperature="67"/>
13        <Day Day="5" Temperature="79"/>
14        <Day Day="6" Temperature="81"/>
15        <Day Day="7" Temperature="79"/>
16        <Day Day="8" Temperature="79"/>
17        <Day Day="9" Temperature="72"/>
18        <Day Day="10" Temperature="45"/>
19        <Day Day="11" Temperature="51"/>
20        <Day Day="12" Temperature="65"/>
21        <Day Day="13" Temperature="74"/>
22        <Day Day="14" Temperature="74"/>
23        <Day Day="15" Temperature="71"/>
24        <Day Day="17" Temperature="59"/>
25        <Day Day="18" Temperature="47"/>
26        <Day Day="19" Temperature="39"/>
27        <Day Day="20" Temperature="66"/>
28        <Day Day="21" Temperature="59"/>
29        <Day Day="22" Temperature="42"/>
30        <Day Day="23" Temperature="41"/>
31        <Day Day="24" Temperature="59"/>
32        <Day Day="25" Temperature="50"/>
33        <Day Day="26" Temperature="63"/>
34        <Day Day="27" Temperature="71"/>
35        <Day Day="28" Temperature="76"/>
36        <Day Day="29" Temperature="76"/>
37        <Day Day="30" Temperature="76"/>
38        <Day Day="31" Temperature="66"/>
39      </Month>
40      ...
41    </Year>
42    ...
43  </AverageDailyTemperature>
44 </WeatherData>

```

Figure 4.16: Public Model State

take. (The X values are in seconds “since” 12:00am January 1, 1970. They are

```
1 <ViewSpecification>
2   <Graph>
3     ...
4     <Point X=-278870400 Y=69/>
5     <Point X=-278784000 Y=44/>
6     <Point X=-278697600 Y=57/>
7     <Point X=-278611200 Y=67/>
8     <Point X=-278524800 Y=79/>
9     <Point X=-278438400 Y=81/>
10    <Point X=-278352000 Y=79/>
11    <Point X=-278265600 Y=79/>
12    <Point X=-278179200 Y=72/>
13    <Point X=-278092800 Y=45/>
14    <Point X=-278006400 Y=51/>
15    <Point X=-277920000 Y=65/>
16    <Point X=-277833600 Y=74/>
17    <Point X=-277747200 Y=74/>
18    <Point X=-277660800 Y=71/>
19    <Point X=-277488000 Y=59/>
20    <Point X=-277401600 Y=47/>
21    <Point X=-277315200 Y=39/>
22    <Point X=-277228800 Y=66/>
23    <Point X=-277142400 Y=59/>
24    <Point X=-277056000 Y=42/>
25    <Point X=-276969600 Y=41/>
26    <Point X=-276883200 Y=59/>
27    <Point X=-276796800 Y=50/>
28    <Point X=-276710400 Y=63/>
29    <Point X=-276624000 Y=71/>
30    <Point X=-276537600 Y=76/>
31    <Point X=-276451200 Y=76/>
32    <Point X=-276364800 Y=76/>
33    <Point X=-276278400 Y=66/>
34    ...
35  </Graph>
36</ViewSpecification>
```

Figure 4.17: View Specification

negative because $1961 < 1970$.)

The next step is for the Projection Function (7) to compute a projection (8). That projection might look something like Figure 4.18. Note the following:

- There’s a suspiciously low reading on the left side of the graph. We’ll deal with

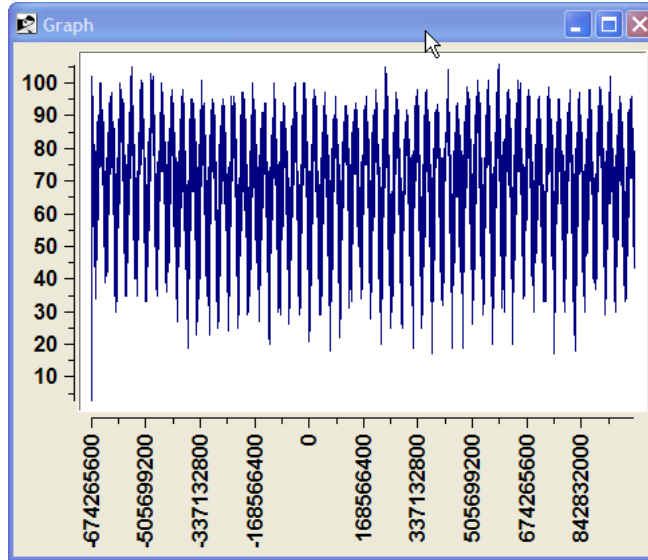


Figure 4.18: Projection

that later.

- There's probably too much data on the graph. The graph has one point for every day between August of 1948 and December of 2001. We'll deal with that later, too.
- The graph could use some labels to make it easier to tell what the data means. We'll deal with that one now.

The labels we will add are of two forms. Some, like date labels for the X axis, come from the model. For these, we will just be projecting this data in a different format. Others, such as descriptive titles for the X and Y axes and the graph itself, are “syntactic sugar”, because they do not come from the model. (This information *could* be in the model, but it isn't in ours.) We will now change the View Function to add these labels to the View Specification. The resulting View Specification is Figure 4.19, and the resulting projection is Figure 4.20.

```

1 <ViewSpecification>
2   <Graph Title="Chapel Hill Maximum Daily Temperatures">
3     <Axis Axis="X" Title="Date" Unit="Time"
4       TickFormat="%Y-%m-%d"/>
5     <Axis Axis="Y" Title="Degrees Fahrenheit"/>
6     ...
7     <Point X="-278870400" Y="69"/>
8     <Point X="-278784000" Y="44"/>
9     ...
10  </Graph>
11 </ViewSpecification>

```

Figure 4.19: View Specification with Labels

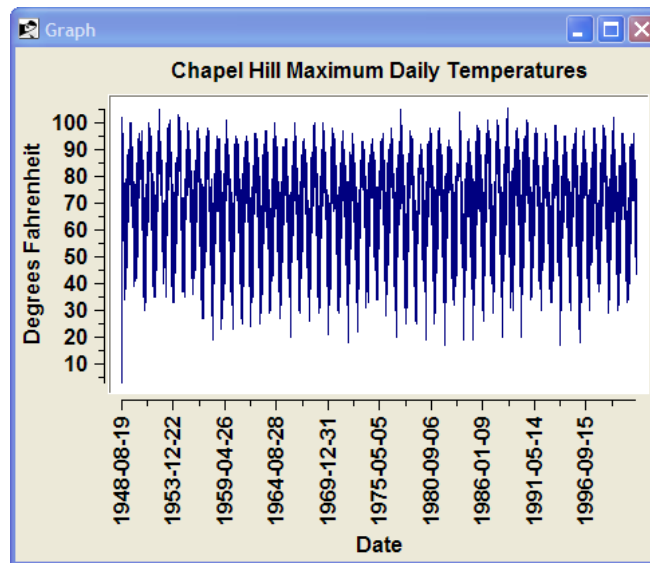


Figure 4.20: Projection with Labels

Now we'll deal with the problem of having too much data on one graph. What we need is to be able to specify a range of dates. This is a good job for a Perspective (10). Figure 4.21 will do the trick. This perspective is requested by the View Function and

```
1 <DateRange>
2   <Start Date="1948-08-01"/>
3   <End Date="2001-12-31"/>
4 </DateRange>
```

Figure 4.21: Perspective

initialized by the Controller (9). Now the View Function will take not only the model as input, but also the perspective. But the View Specification still hasn't changed, since the Start and End times cover the entire data range.

We'd like to figure out why there's an abnormally low reading toward the left of the original graph. To do that, we need to zoom in on that part of the graph by adjusting our Perspective. But how do we do that? The View Function must provide a UI so that the user can adjust the range of data being displayed. We will add two sliders to the ViewSpecification so that we can specify the start and end values for the Perspective (Figure 4.22). Figure 4.23 shows the new projection. The Controller will register an interest in slider events. It can register this interest once at the ViewSpecification node of Figure 4.22 (because events bubble up), or it can register its interest separately at each of the Slider nodes.

Now we can move the sliders to show the left of the graph in more detail. This will apply events to the Slider nodes of the View Specification. The controller will respond to these events by changing the Perspective values, and the View Function will change the View Specification accordingly, as shown in Figure 4.24. (Note the lack of ellipses, indicating that the points explicitly shown are the only ones to be

```

1 <ViewSpecification>
2   <Graph Title="Chapel Hill Maximum Daily Temperatures">
3     <Axis Axis="X" Title="Date" Unit="Time"
4       TickFormat="%Y-%m-%d"/>
5     <Axis Axis="Y" Title="Degrees Fahrenheit"/>
6     ...
7     <Point X="-278870400" Y="69"/>
8     <Point X="-278784000" Y="44"/>
9     ...
10  </Graph>
11  <Slider Title="Start" Min="-675878400" Max="1009785600"
12    Value="-675878400"/>
13  <Slider Title="End" Min="-675878400" Max="1009785600"
14    Value="1009785600"/>
15 </ViewSpecification>

```

Figure 4.22: View Specification with Sliders

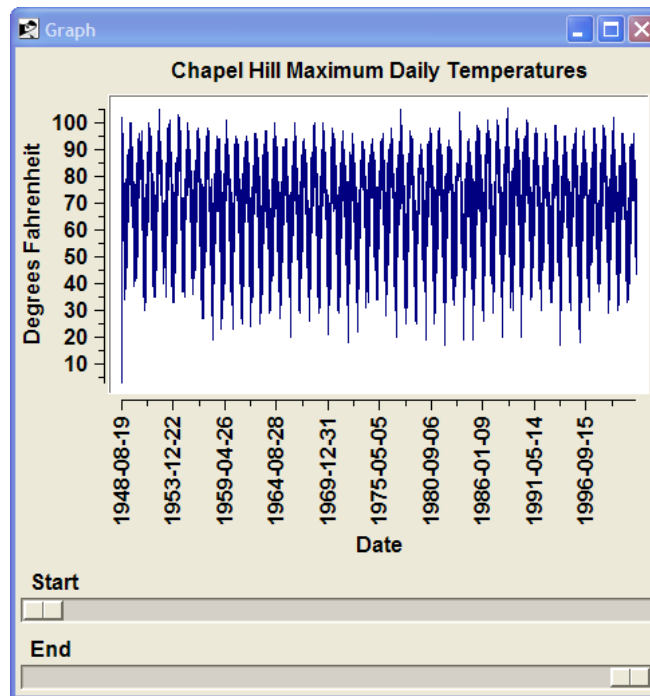


Figure 4.23: Projection with Sliders

```

1 <ViewSpecification>
2   <Graph Title="Chapel Hill Maximum Daily Temperatures">
3     <Axis Axis="X" Title="Date" Unit="Time"
4       TickFormat="%Y-%m-%d"/>
5     <Axis Axis="Y" Title="Degrees Fahrenheit"/>
6     <Point X="-675878400" Y="90"/>
7     <Point X="-675792000" Y="87"/>
8     <Point X="-675705600" Y="90"/>
9     <Point X="-675619200" Y="82"/>
10    <Point X="-675532800" Y="86"/>
11    <Point X="-675446400" Y="85"/>
12    <Point X="-675360000" Y="82"/>
13    <Point X="-675273600" Y="84"/>
14    <Point X="-675187200" Y="87"/>
15    <Point X="-675100800" Y="89"/>
16    <Point X="-675014400" Y="89"/>
17    <Point X="-674928000" Y="88"/>
18    <Point X="-674841600" Y="92"/>
19    <Point X="-674755200" Y="95"/>
20    <Point X="-674668800" Y="87"/>
21    <Point X="-674582400" Y="85"/>
22    <Point X="-674496000" Y="85"/>
23    <Point X="-674409600" Y="89"/>
24    <Point X="-674323200" Y="91"/>
25    <Point X="-674236800" Y="86"/>
26    <Point X="-674150400" Y="87"/>
27    <Point X="-674064000" Y="90"/>
28    <Point X="-673977600" Y="90"/>
29    <Point X="-673891200" Y="92"/>
30    <Point X="-673804800" Y="94"/>
31    <Point X="-673718400" Y="98"/>
32    <Point X="-673632000" Y="99"/>
33    <Point X="-673545600" Y="102"/>
34    <Point X="-673459200" Y="3"/>
35    <Point X="-673372800" Y="96"/>
36    <Point X="-673286400" Y="92"/>
37  </Graph>
38  <Slider Title="Start" Min="-675878400" Max="1009785600"
39    Value="-675878400"/>
40  <Slider Title="End" Min="-675878400" Max="1009785600"
41    Value="-673459200"/>
42 </ViewSpecification>

```

Figure 4.24: View Specification with Changed Slider Values

graphed; all others have been removed from the View Specification of Figure 4.22.) The View Function can also now narrow its interest in the Public Model State, which means that less data needs to be cached in the Model Cache. Finally, the Projection Function will respond by zooming in the graph as shown in Figure 4.25.

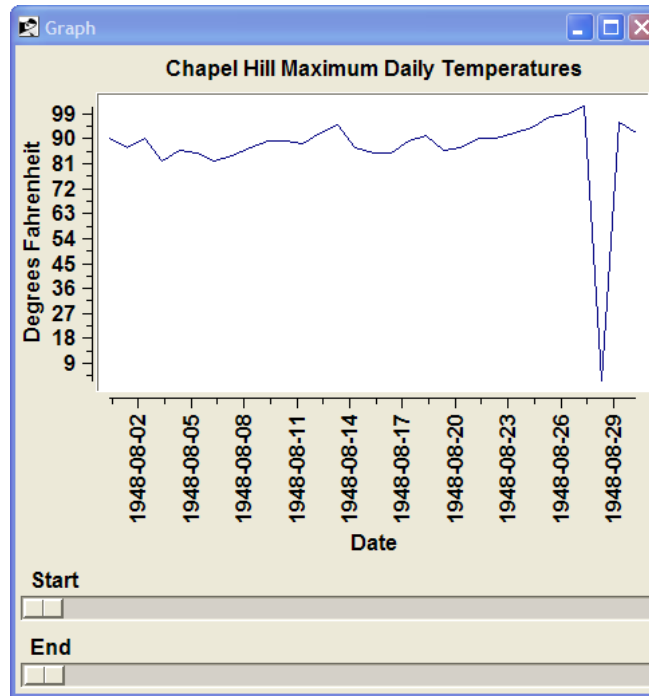


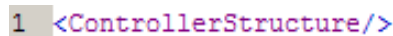
Figure 4.25: Zoomed Projection

Now we can see that there's anomalous data somewhere around 1948-08-28. How can we get more detail? One way would be to write a new View Function that would take the same Public Model State as input and display it in XML, similar to Figure 4.16, instead of graphically, like Figure 4.25. This would give the user the same insight we have in this paper by exposing the underlying data structures on his display as we can in our figures. This same View Function could be very handy for debugging in general; it could be applied to any Public Model State, ViewSpecification, Perspective, Controller Structure, etc. Using such a projection, we could see

that the anomalous data is actually on 1948-08-29, where a high temperature of 3 degrees Fahrenheit follows a high on the 28th of 102 degrees.

Suppose now that we have permission to edit the original model data. The controller would need to register for events on the Graph node of Figure 4.24. (The controller is a simple state machine because it needs to act on the Controller Structure based on transient events and sequences of events (gestures)). Some event (e.g., Control-leftclick) or series of events (e.g., right-click and select Delete from a menu) could mean “delete the point under the cursor”. Similarly, we could define a series of events that could move the point under the cursor. The controller would respond to these events and then manipulate the Controller Structure (11) to request that the point be deleted or moved.

There are numerous ways to set up the Controller Structure. For example, one might have one node per point the graph; in this case one could send a Delete event to that node to delete it, or a Move event to move it. For this example, I have taken a different approach (Figure 4.26), which is to define a controller consisting of a single node, through which more complex events can be sent⁶. A “DeletePointAt



```
1 <ControllerStructure/>
```

Figure 4.26: Controller Structure

-673459200” event can then be sent to the ControllerStructure node of Figure 4.26. This event would propagate to a similar node in the Aggregated Controller Structure (12). The Model would listen for such events, and respond by deleting the point from the database and the Public Model State. This change would be propagated to the Model Cache, and the View Function would respond by updating the View

⁶Operations on the model can be requested by either applying events to a Controller Structure or modifying the Controller Structure.

Specification. Finally, the Projection Function would react by updating the graph as shown in Figure 4.27.

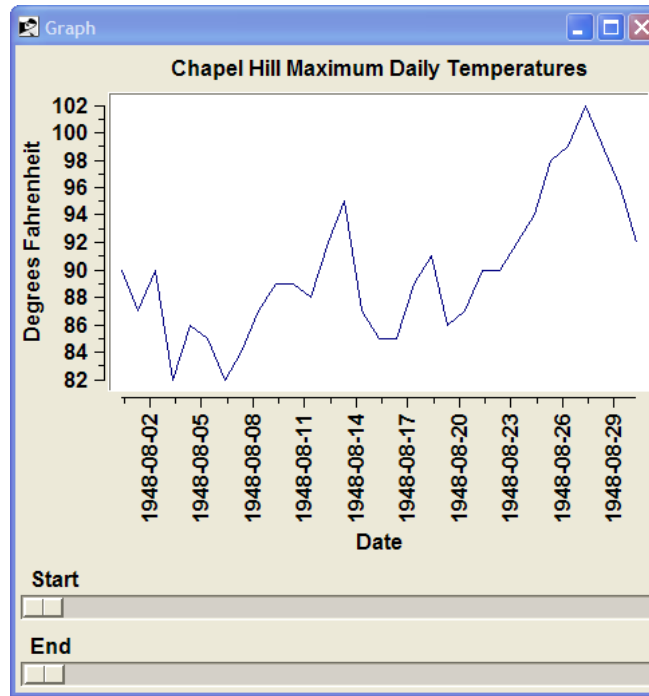


Figure 4.27: Projection after Deleting a Point

So far we have only addressed the single-user case; now we need to consider collaboration. Suppose Client 2 now joins the conference as a latecomer. Let's assume for now that it requests the same View Function and the same inputs to that function as Client 1; i.e., that the conference is WYSIWIS. The View Function (14) is downloaded and the Model Cache (13) is set up and populated. A slave Perspective (19) is created and linked to the central slave (21). The View Function generates the View Specification (15). The Perspective Function (16) is instantiated, and it generates the projection (17) of Figure 4.27. All of this is a straight-forward repetition of what happened for Client 1, except that the Perspective is a Slave, not a Master. If Client 2 has permission to interact with the application, a Controller (18) is set up, and linked

to the Perspective Slave. The Controller generates the Controller Structure, which is linked to the Aggregated Controller Structure (13). The only interesting point here is that the events applied to the Controller Structures of both clients are serialized when as they are forwarded to the Aggregated Controller Structure. I am assuming that operations performed on the model in response to these events are atomic, or that the events invoke a transaction mechanism to achieve transaction isolation among clients. The Perspective Master role can migrate between the clients, depending on the semantics required by the application, which must be communicated to the Concur infrastructure.

It should be apparent from the above discussion that migration is a simple matter. One's client instantiations are torn down and reproduced elsewhere. It should also be apparent that there may be times when there are zero clients, and that this situation is handled gracefully.

Now we will discuss the issue of divergence. Consider the following possibilities:

- Client 2 wishes to look at a different date range for the data. This is accomplished by giving Client 2 its own perspective. Note that the Model Cache components will then differ for the two clients.
- Client 2 is communicating with Client 1, so they want their perspectives held in common. In the process, however, Client 2 decides to look at the Minimum Daily Temperature data along with the Maximum Daily Temperature data. This is accomplished by adding the Minimum data to the Client 2 View Function's interest set⁷.
- Client 2 is accustomed to a different set of key bindings for interaction with the

⁷The View Specification Syntax would need to be enhanced to support multiple lines per graph; this is easily accomplished.

application. (Think emacs vs. vi.) This is accomplished by swapping out the controller.

- Client 2 wishes to see the data in bar chart form (Figure 4.28). This is accomplished by swapping out the View function.

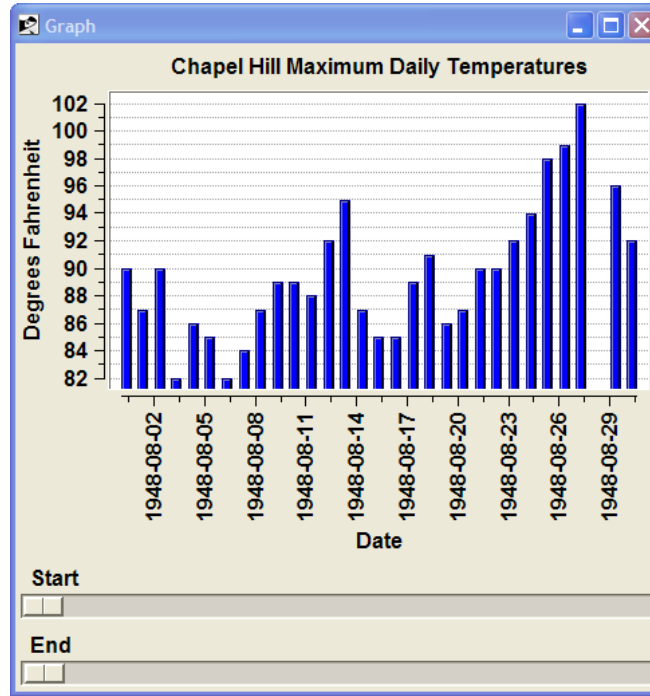


Figure 4.28: Projection in Bar Chart Form

These and other divergence scenarios are reasonably simple to accommodate using the Concur architecture.

But what mechanism would be used for identifying the View Functions, their Public Model State and Perspective inputs, and Controllers? While I have given a fair amount of thought to this, it is beyond the scope of this dissertation. I envision something like putting the controller into meta-mode, where interactions are used to edit the user interface by choosing different components, and then returning to

normal mode, where interactions invoke operations on the application. These and other ideas for future research are discussed in Chapter 7.

4.4 Debugging, Testing, and Scripting

I will conclude this chapter by offering a few paragraphs on how the Concur architecture facilitates debugging, testing, and scripting in ways that are a dramatic improvement over the state of the art.

I am a strong believer in the following sometimes touted but rarely followed software engineering principle:

Every interactive software application should be built such that all application operations can be invoked via a script (Figure 4.29). User interfaces should then be built such that they invoke all operations via the scripting layer.

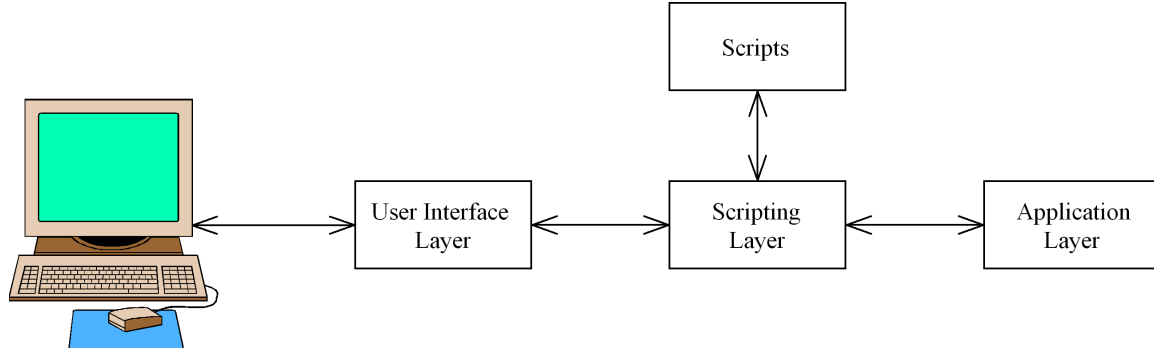


Figure 4.29: Software Architecture with a Scripting Layer

This principle ensures, among other things, that applications can be automated and that extensive regression tests can be developed. Since so much of today's software lacks such a scripting layer, testing is often done using software that emulates user inputs delivered to the actual user interface. This mechanism is non-deterministic

(e.g., timing dependent) and sensitive to minor changes in the user interface; it is therefore not robust. It is also difficult to capture the results of an operation (i.e., its impact on the user interface) and compare them to expected results, especially in the presence of small UI changes.

Concur's use of a scriptable data representation API supports the scripting of applications at multiple layers. Data representations using this API are utilized as a communication mechanism among software components. Communication consists of the creation and modification of hierarchical data structures and the application of events to those structures. The rectangles in Figure 4.14 are such data structures, and the ovals (continuously evaluated functions) and rounded rectangles (state machines) are such components. The use of continuously evaluated functions, which contain no essential state internally, ensures that all the important state is accessible externally to components. The use of a scriptable API with event and notification capabilities ensures that the data structures between components, which do contain essential state, can be observed and manipulated by debugging, test, and automation software. The end result is that everything in the system can be tested, observed, and automated deterministically via scripts (upon which UIs can be built), except for the Projection Function and its results, and the Model's external references. The Projection Function (which is actually a state machine with functional guarantees) is a small component that can be tested separately and reused for all applications, so it can be trusted to be correct.

As examples, consider the following capabilities supported by Concur:

- Communication between components can easily be captured and analyzed or replayed.
- A simple software component can manage the Model Cache to emulate a Model

before the Model is available, or in software (e.g., test) environments that don't support the full Model. View Functions can be tested by simulating a Model Cache and programmatically observing the View Specification.

- View Specifications can easily be dumped as a compact, definitive, and structured (analyzable) alternative to pixel dumps. They can be used to visually reconstruct projections.
- The ability to capture and replay the dynamic visual history of a session comes automatically.
- A simple software component can apply a scripted set of events to a View Specification, and the changes to the View Specification resulting from the Controller/Model/View Function loop can be programmatically analyzed.
- A simple software component can apply events or structural changes to a Controller Structure or Aggregated Controller, and the Public Model state can be programmatically analyzed, to test Model behavior.
- Special View Functions can be created to enable one to observe any of the data structures in the system. Similarly, special Controllers can be created to manipulate these structures. These special View Functions and Controllers can be application-independent, and can therefore be used with any application.

Chapter 5

Concur Implementation

In this chapter I will give a brief tour of the implementation of the Concur proof of concept infrastructure. First I will discuss the programming language and libraries used, then the class library undergirding the server and client. Finally, I will describe the server and client software themselves. I will only discuss the most important classes of each component, and these only at a high level of abstraction. Applications will be addressed in the next chapter.

5.1 Programming Language and Libraries

My programming language choice was based on the following requirements:

- Because Concur relies heavily on distributing code among hosts, it was convenient and practical to choose an interpretive language.
- Distributed client code needs to be run in a secure container. I therefore needed a language with security mechanisms appropriate for building such a container.
- Interpretive languages can become unwieldy unless they have object-oriented capabilities, so these were desirable as well.

- Since performance was a central concern of my thesis, it seemed prudent to choose an interpretive language with reasonably good performance characteristics, and which offered the ability to incrementally and transparently replace components written in an interpretive language with functionally-equivalent ones written in a compiled language.
- I also needed easy access to user interface capabilities and an API for representing hierarchical data structures.
- It was desirable for the language chosen to be available on the three major operating systems (Microsoft Windows, UNIX variants, and Apple OS/X).
- Finally, I wanted a robust environment on which to build, so that I could expend my efforts on the proof of concept itself.

The language I chose was IncrTcl[Smi00], which meets all of the above requirements. IncrTcl is an object-oriented extension to Tcl[Ous93][WH03]. All the important object-oriented language facilities are supported, using keywords and syntax similar to those of C++[Str00]. Method bodies are written using the Tcl language.

Tcl itself, which underlies IncrTcl, is a simple, consistent, and robust interpretive language developed by John Ousterhout at the University of California at Berkeley. It is easily embedded in programs written in other languages, and it contains an extension facility that allows command libraries to be written either in Tcl or in a compiled language (e.g., C++, Java[AGH05], or C#[Lib03]). Tcl also performs quite well, for an interpreted language. This is because the interpreter contains a on-the-fly byte code compiler and supports dual-ported objects.

Dual-ported objects are a mechanism for maintaining the value of an object both as a string representation (as everything is specified in scripts and in I/O) and as

some binary representation (e.g., an integer). For example, suppose a Tcl variable containing the string representation of an integer is passed to a Tcl command requiring an integer. The string is converted to a binary integer which is stored alongside the string as an alternate representation. If, as is likely, the variable is again used as an integer, its integer representation is still available. If it is needed as a string, the string representation is also available. If it is then needed as a third type (e.g., float), the integer representation is discarded and replaced by the new representation. Of course, if one representation is changed (e.g., the integer is incremented), the other is invalidated and must be re-created on demand. Tcl also supports sharing of objects with the same value, with copy-on-write semantics. Together, the byte code compiler, dual-ported objects, and shared object values combine to make Tcl an efficient interpretive language.

Tcl also provides native capabilities for running scripts in a safe interpreter with limited and precisely-specified capabilities. Safe interpreters can be set up to make trapped calls (analogous to system calls on an operating system) that are executed in a privileged master interpreter. Concur makes use of safe interpreters for executing application code.

In addition to IncrTcl, there are many other Tcl extensions available. The basic user interface extension, Tk (also developed by John Ousterhout), is nearly as simple and fully as robust as Tcl, but it does not provide all the user interface components one might want. The UI needs of the Concur prototype infrastructure are minimal, so Concur only uses one other Tcl UI extension, an image management extension called Img[Nij00]. A production implementation of Concur would use many other UI APIs on various platforms for its projection functions. Concur also needed an API with an event mechanism for managing hierarchical data structures. Originally the Document

Object Model (DOM) API[Mar02] was used¹, but I eventually determined that the DOM did not have the right facilities for caching and replicating data structures. I then turned to a hierarchical data structure API that was part of the BLT[How97] extension to Tcl, and I wrote an IncrTcl wrapper around it to augment it with the facilities I needed.

5.2 Concur Class Library

The Concur class library contains utility classes that are useful to both the client and server processes. The most important of these are described in this section.

The **ConcurSafeInterp** class implements a restricted container that only allows certain Tcl commands to be executed. Some of these commands trap out to a privileged interpreter, much like system calls do in an operating system. Methods are provided that allow a privileged interpreter to determine which commands can be executed and how they may trap to a privileged interpreter.

The **ConcurChannel** class represents an end-point of a TCP/IP connection. Interaction among processes in Concur is normally asynchronous (message-based), not synchronous (remote-procedure-call-based). Any synchronization that might be required is implemented above the **ConcurChannel** level. Thus the ConcurChannel class is oriented around sending and receiving messages. Each message sent is a Tcl command intended to be executed on the other side of the link. The **Send** method packages up a command and writes it to its end of the link. It will optionally delay the message by a specified amount of time, to simulate network delays. The **Receive** method reads data off its link, parses it into commands, and executes these in a safe

¹To use the DOM API, I wrote a Tcl program that generated a wrapper (written in Java) for the Xerces-J DOM API[Xer08], as a Tcl extension.

interpreter. Each channel is given an instance of a sub-class of the **ConcurChannelCommands** object, which defines which commands it is allowed to execute for that channel. This is how all communication among processes in Concur occurs.

The **ConcurProcess** class is a super-class providing a runtime environment for the server, client, and model (producer) processes.

The **ConcurEventManager** class implements a process-level event-based communication mechanism. It has methods for registering an interest in and triggering events. This allows components within a process to communicate event information with each other indirectly. That is, the component triggering an event need not know about components interested in the occurrence of that event.

The **ConcurTree** class represents a hierarchical data structure. It is used to represent models, perspectives, and view specifications². It is also used to encapsulate view functions and controllers, by including Tel code in a tree data structure. This allows the same tree replication facilities used for models and perspectives to be used for distributing view functions and controllers. Representing code in a tree also provides a convenient mechanism for view functions and controllers to access associated data, as they are allowed to access data items in the tree containing their code.

ConcurTree instances can be named and referenced using a unique path string, as are files in a file system. This, for example, is how models and perspectives are identified. They are also given a shorter unique ID, which reduces network traffic and improves performance. (File systems do the same thing.) Particular nodes of a tree data structure can also be referenced by a path from the root to the desired node,

²A production implementation of Concur would provide a **ConcurDAG** class, representing a Directed Acyclic Graph (DAG). None of the example applications written for this dissertation needed DAG generality.

and by an ID that is unique to within the tree.

Each node in a tree can have any number of attributes (name/value pairs) associated with it. All necessary facilities for traversing trees, reading or writing node attributes, and changing the structure of a tree are implemented as **ConcurTree** methods.

The **ConcurTree** class also provides facilities implementing Subject behavior in the push Subject/Observer paradigm. That is, an observer can register an interest in specific kinds of changes that might occur in a tree (e.g., structural changes or attribute creations, deletions, or modifications). These notifications bubble up from the nodes where the changes occur to the root, so that one can register an interest in all the nodes of a sub-tree by registering an interest at the root of the sub-tree. A registration request contains a Tcl command that will be executed with additional parameters describing the change, when a specified change occurs.

Replication, master/slave behavior, and migration behavior is mostly implemented in the **ConcurTree** class. Master or slave status can be applied to a sub-tree. If a change to a node of a local instance of a tree is requested, the tree can determine whether or not it is the master for that node. If it is, the change is made locally and then replicated over the network to other instances of the tree. If the local tree is a slave for that node, it simply forwards the request toward the master instance. The migration algorithm (changing the master of a sub-tree) is also implemented mostly in the **ConcurTree** class. This algorithm queues requests in order to ensure that operations are not lost or reordered during a migration.

As part of its replication facility, the **ConcurTree** implements partial caching of sub-trees. That is, if an observer wishes to have a replica of only part of an existing tree, it can specify such in its registration request. For example, an observer can specify that it only wishes to cache a particular subject sub-tree to a given depth.

In this way, an entire model needn't be replicated if a view function currently only needs access to a portion of the model.

The **ConcurTree** class implements some higher-level attribute modification logic that essentially implements typing of attributes. For example, integer and vector operations are supported. Typed attribute change notifications are also implemented, which allows modifications to attributes to be described succinctly in a notification. For example, inserting a character into a large block of text can cause a brief description of the change to be sent in a notification, instead of re-sending the entire block of text.

A transaction mechanism is also implemented in **ConcurTree**. This allows multiple changes to be made atomically by bracketing them with **BeginTransaction** and **EndTransaction** method calls. Any change notifications that occur during a transaction are queued until the end of the transaction. When trees are replicated, transaction boundaries are passed along to the replica (i.e., the change notifications are also bracketed). In this way we can ensure that view functions do not receive invalid, transient inputs (i.e., inputs outside of their domain). It also keeps projection functions from generating distracting transient UI modifications.

Application code and configurations are specified in XML. The **ConcurXML** class is capable of reading an XML file and converting it into a **ConcurTree** instance, making it accessible to the Concur infrastructure and/or applications. A production version of Concur would also implement a mapping from **ConcurTree** to XML, e.g., for taking a snapshot of a view specification.

The entire Concur library consists of about 2850 lines of code, 600 of which is for performance monitoring classes. The heart of the Concur library is the **ConcurTree** class, which consists of about 1300 lines of code.

5.3 Server Process

The server process code consists of a Tcl script of about 200 lines of code. It launches producers (model applications) accepts connections from clients and producers, and loads and distributes configuration information and code for perspectives, view functions, and controllers. It also hosts **ConcurTree** instances for models, perspectives, and aggregated controllers. These tree instances replicate themselves as necessary.

5.4 Client Process

The client process is more complicated than the server process because it includes projection function classes, and abstract super-classes that are sub-classed by application code. The super-classes used by application code create containers in which models, perspectives, controllers, and view functions can run. The classes associated with the client process are described briefly below. In all, the client-specific classes and script consist of about 1000 lines of code.

The **ConcurClient** class creates a runtime environment for clients and producers. It consists mostly of code to connect to the server and to shut down client processes. The **ConcurViewer** class is a sub-class of **ConcurClient** that is specific to processes hosting view functions and projections (i.e., not producers). It is responsible for requesting model and perspective replicas, view functions, and controllers from the server and for setting up safe containers for application code.

The **ConcurPlaypenCommands** class defines commands that can be executed by view functions and controllers in a safe container. The **ConcurClientChannelCommands** class does the same for the safe client-side channel interpreter.

The **ConcurProjection** class is a super-class for all projection function classes. Each projection consists of an instance of a **ConcurProjectionRoot** class instance associated with a view specification. The **ConcurProjectionRoot** class looks for sub-trees of the view specification corresponding to particular UI technologies and instantiates UI-specific projection functions for each of these sub-trees.

The Concur prototype currently supports only one UI technology, Tk. An instance of the **ConcurProjectionTk** class maps a view specification sub-tree of type Tk to the actual Tk code that maintains a particular Tk projection on the screen. That is, it traverses an initial view specification, creating the specified user interface (projection), and then monitors the view specification for changes and updates the projection accordingly. It also sets up UI event bindings for controllers and maps low-level Tk events applied to the projection to events applied to the view specification. These events, in turn, are observed by controllers, which map them to operations on models and perspectives. The **ConcurProjectionTk** class accounts for about 600 of the 1000 client-specific lines of code. This is effort that needs to be duplicated for each UI technology.

Chapter 6

Analysis and Evaluation of Concur

In Sections 1.9 and 4.1 there are two partially overlapping lists against which the Concur prototype described in Chapter 5 can be evaluated. In this chapter I will combine these lists and evaluate the Concur prototype with respect to each of the items in the new list. The next section will present these criteria for analysis.

6.1 Criteria for Analysis

In Section 1.9 a list of the contributions of this dissertation was enumerated. This list consists of items that are new in this work. In Section 4.1, a list of requirements for Concur was enumerated. This list contains both items that are new in this work, and items that may already have been demonstrated by others, but that Concur must also achieve.

The following list combines and summarizes the above two lists into the items against which Concur will be evaluated in this chapter.

1. Section 1.9 states that one of the contributions of this work is a novel classification of application software components (entities) based on their migration

characteristics. This taxonomy was presented in Section 3.4. Migration times must be better than 100ms where network delays allow.

2. I will show that these entity types can be used to build a wide range of applications by building three sample applications chosen to be representative of applications in general.
3. By means of analyzing the above sample applications, I will argue that Concur provides a reasonable and understandable programming environment based on a centralized architecture and collaboration-unaware applications. I will demonstrate that this architecture does not require model synchronization algorithms akin to those used in replicated architectures, which simplifies the programming of the infrastructure and/or applications. I will also argue that the kinds of user mental model complexities required by a replicated architecture are not present in this architecture.
4. I will argue that Concur is deterministic.
5. I will argue that above-mentioned entities can be used to implement a wider range of desirable per-participant divergence scenarios than existing systems.
6. I will argue that Concur supports individual work as well as all four classes of collaborative work illustrated in Figure 1.1, and that Concur supports transitions among all these forms of work. In the process, I will show how Concur supports latecomers and mobility.
7. I will deliver a prototype infrastructure supporting these applications, built as a multi-centered centralized system with entity migration support.
8. I will demonstrate that this infrastructure and applications conforming to it

give substantially better interactive performance to all participants than does a purely centralized architecture, and close to that of a replicated architecture. I will demonstrate that this architecture scales better in terms of both processor and network bandwidth utilization than a purely centralized architecture.

9. I will demonstrate the advantages of predictive migration based on telegraphed user intentions, made possible by Concur's fast migration times, rather than a past history of interaction.

Some of the above criteria will be met via informal argument, while others will be demonstrated using the results of formal experiments. The next two sections will describe the experimental environment and the experiments themselves.

6.2 Experimental Environment

A diagram of the experimental environment was shown in Figure 1.24. It consisted of two 1Gb/s Ethernet switches joined using routers connected by 1Gb/s links. On one of these switches was a single server host configured as follows:

- 3 GHz Pentium D dual-core processor
- 1 GB RAM
- 1 Gb/s Network Interface Controller (NIC)
- Red Hat Linux release 4

The other switch contained six client hosts configured as follows:

- 400 MHz Pentium II processor
- 256 MB RAM

- 100 Mb/s NIC
- FreeBSD UNIX version 5.4

The Concur infrastructure described in Chapter 5 ran on these hosts. A Server process (which served as the physical center of the network) and a Producer process (which managed the model data) ran on the server host, while a Client process and an X server ran on the client hosts. Client and Producer processes communicated with the Server process via TCP/IP[Com00] sockets.

Note that by today's standards, these client hosts are severely under-powered. As a reference point, in August of 2008, A Dell XPS 420 desktop computer selling for \$900 is configured as follows:

- 2.4 GHz Intel quad-core processor
- 3 GB RAM
- 1 Gb/s NIC
- Windows Vista Home Premium

Nevertheless, the client computers used for these experiments gave adequate interactive performance for up to 6 users in the collaborative experiments described below. One would expect that the computers of today and the future would have no performance problems at all with the Concur infrastructure.

Eighteen additional hosts were also attached to each of the two Ethernet switches. These hosts were used to generate background traffic for half of the experiments, in order to measure the impact of such traffic on the Concur infrastructure. The traffic generated was a replay of one hour's actual traffic between the UNC network and the Internet, captured on August 3, 2004. (See [HC06] for details on how this traffic was

captured and replayed.) The UNC side of the traffic was replayed on the client switch, while the Internet side was replayed on the server switch. All traffic was directed at the other switch, so that all traffic went through the routers, creating contention for the network. The background traffic was re-sampled such that it generated a full load of realistic traffic over a 90 minute period. Figure 6.1 shows the total background traffic (sum of both directions) for several 90-minute runs. (The data points represent

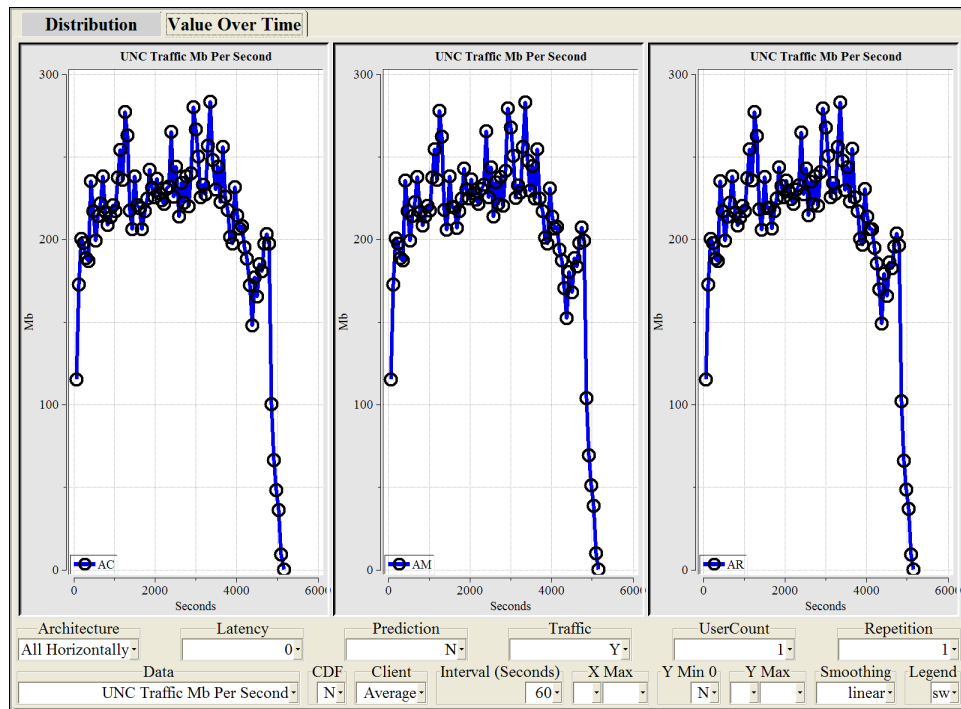


Figure 6.1: Background Network Traffic

averages over 60-second intervals. The traffic is bursty, with measured bursts of up to 600Mb/sec averaged over 10ms intervals.) The consistency of this traffic across runs is apparent from this diagram. Four Concur experiments were run sequentially within each 90-minute replay of this background traffic.

As it turns out, the background network traffic's impact on the Concur experiments was minimal. (For example, see Figures 6.2, 6.3, and 6.4.) (The biggest

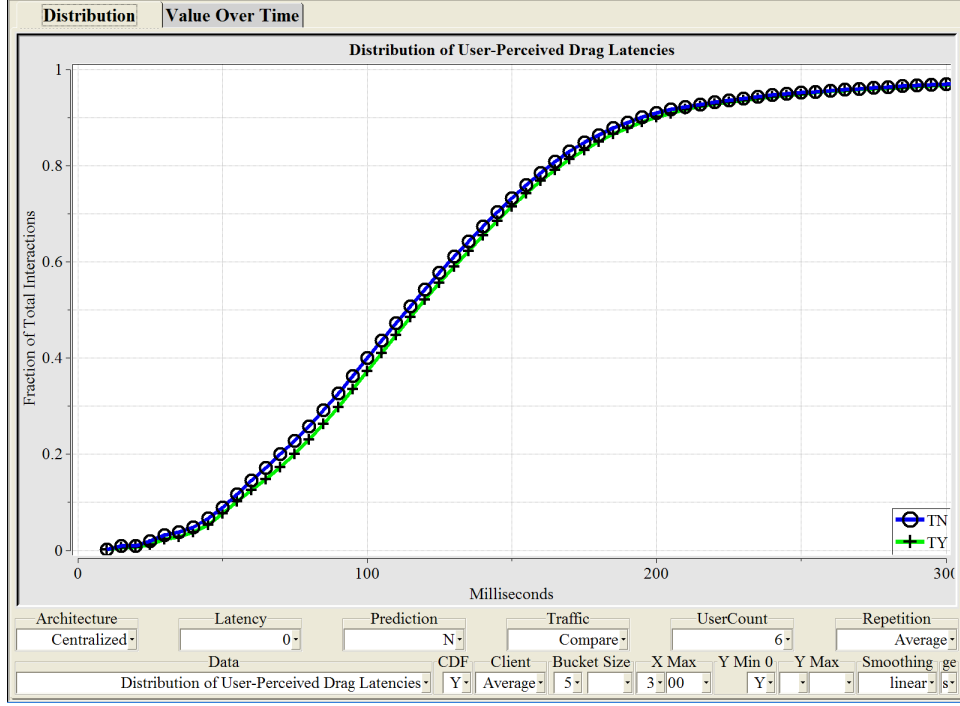


Figure 6.2: Background Traffic Impact on Drag Latency (Centralized)

impact was on the Centralized architecture, because it is the architecture most dependent on the network.) This is a good result for Concur, as it demonstrates that the infrastructure works well under normal network loads. Most of the performance results shown in the remainder of this chapter will show only those experiments without background traffic. Corresponding experiments with network traffic had similar results.

6.3 Experiments Performed

The application used for all experiments was a 70-piece jigsaw puzzle (Figures 6.5 through 6.8). The puzzle application was chosen because it progressed from mostly individual work (Figure 6.6) to mostly collaborative work (Figure 6.7) as the puzzle

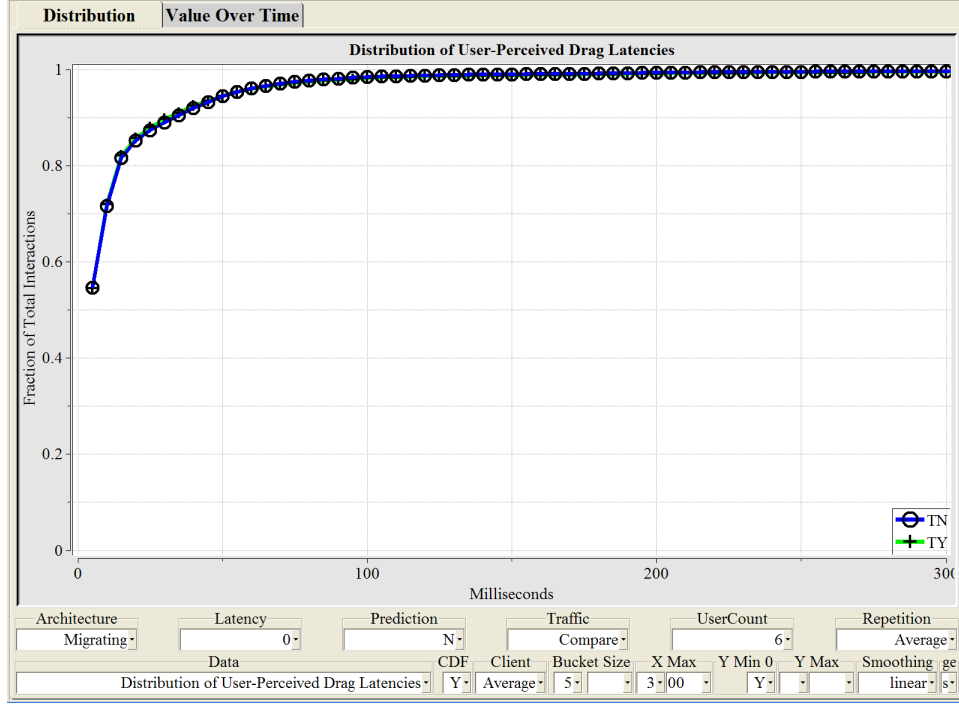


Figure 6.3: Background Traffic Impact on Drag Latency (Migrating)

was solved. Each experiment started with the same puzzle pieces, but their placement was random and different for each experiment. Puzzle pieces were always in their correct orientation; random rotations were deemed an unnecessary complication for the purposes of these experiments.

The user interface for the puzzle application was as follows:

- Moving the mouse cursor over a piece highlighted it with a yellow border.
- Right-clicking on a piece moved it to the bottom of a stack of overlapped pieces.
- Left-clicking on a piece raised it to the top of a stack of overlapped pieces.
- Moving the mouse cursor with the left button depressed dragged the piece. Releasing the left button “dropped” the piece.

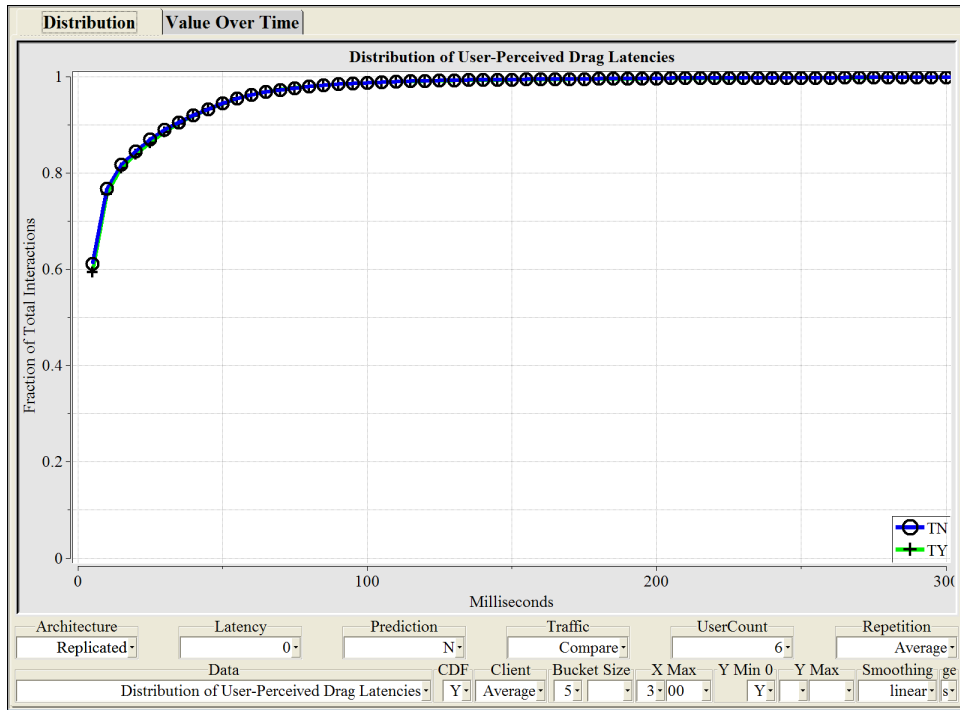


Figure 6.4: Background Traffic Impact on Drag Latency (Replicated)

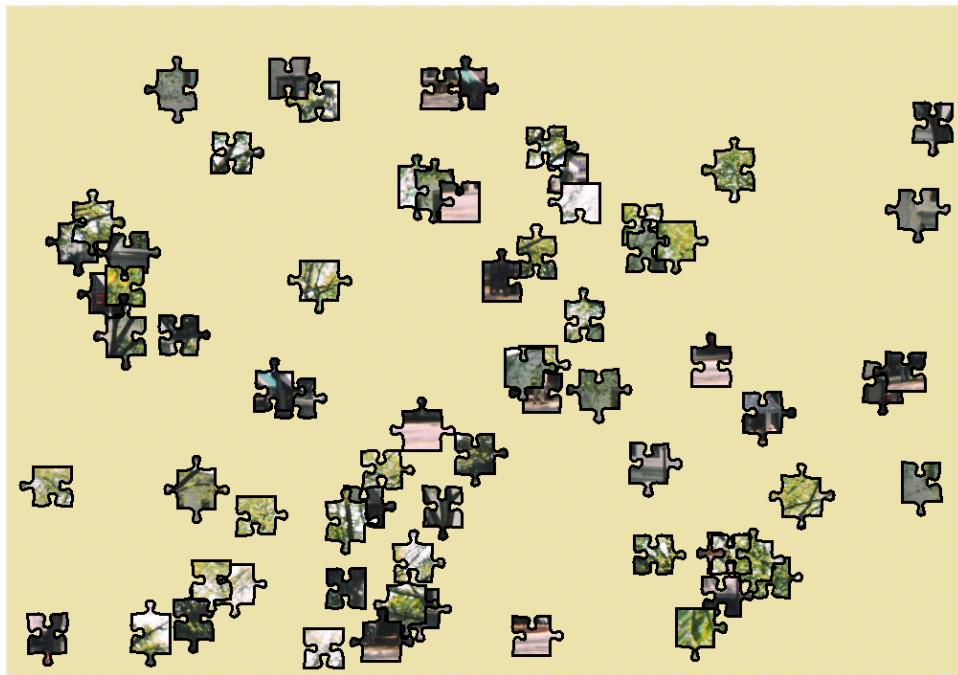


Figure 6.5: Puzzle Starting Point

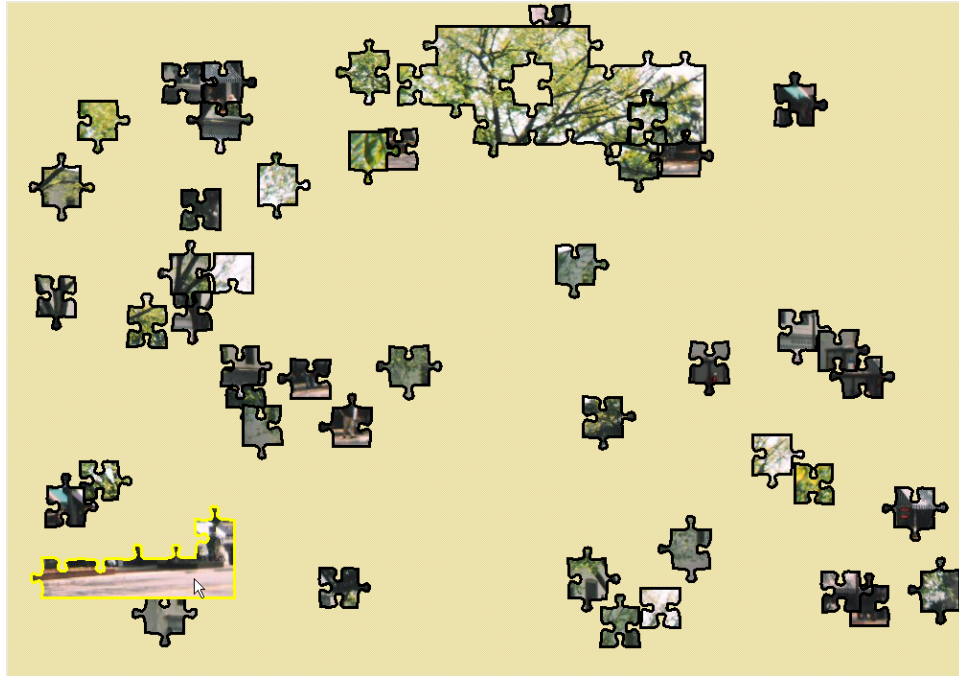


Figure 6.6: Puzzle in Early Stage of Completion

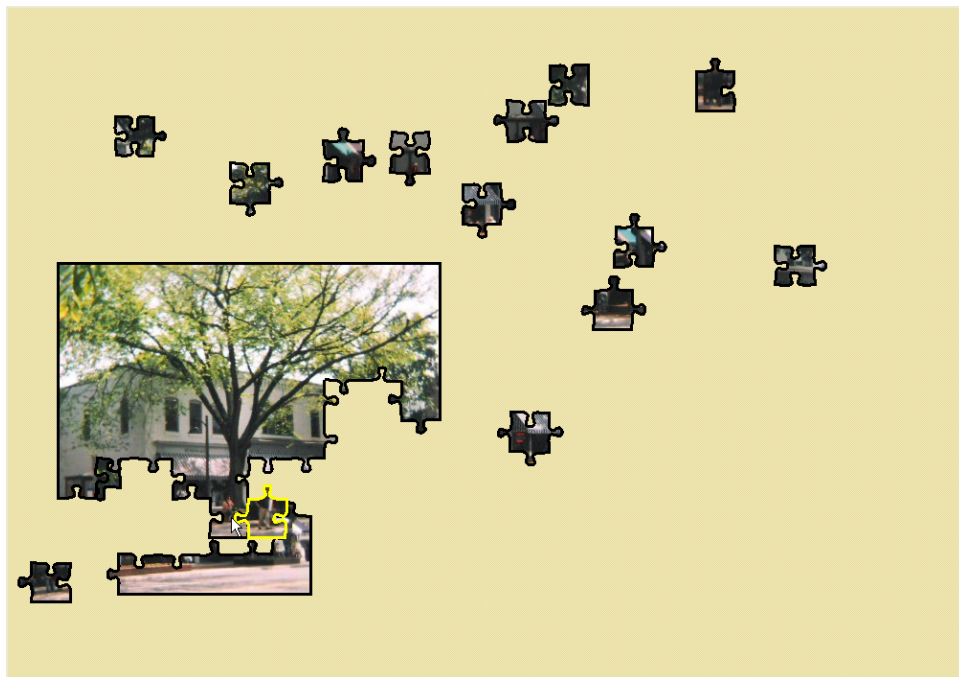


Figure 6.7: Puzzle in Late Stage of Completion



Figure 6.8: Complete Puzzle

- Middle-clicking on a piece snapped it together with any matching pieces within a few pixels of their correct position with respect to the piece under the cursor.

Each client for the experiments additionally contained puzzle solving logic emulating a user's attempts to solve the puzzle. The solver applied the above low-level events to the view, just as a real user would. Think times, attempted matches, and drag speeds simulated those of a real user. The solver algorithm had the following characteristics:

- Each user's solver randomly chose a "favorite" color. It began by attempting to match pieces near that color, and progressed to colors further away from its favorite color as the puzzle solution proceeded. This tends to encourage different solvers to work on different parts of the puzzle.
- The solver would only attempt a match where it made sense. For example, it would only attempt to match the sides of two pieces if the sides were of opposite gender, and if the edge sides aligned properly.
- Priority was given to matching pieces where the sides to be matched were similar in color. The chosen puzzle image, a picture of Franklin Street in Chapel Hill, NC (Figure 6.8), was selected because of its areas of different color.
- Pieces were never moved off the board. A piece's position would be adjusted if a match could not be attempted in its current position without moving the matching piece off the board.
- If the solver decided to stop attempting to match a piece, it was positioned randomly before being dropped. This keeps pieces from piling up in one place.
- Attempts are made to move smaller fragments in preference to larger ones.

- If the solver is moving the cursor toward a piece to be moved and that piece is moved by another player, the solver backs off and tries another match. The same thing happens if the solver is dragging a piece and it is moved by another player, or if it is dragging a piece toward another piece to attempt a match, and the other piece is moved by another player.

These are all characteristics common to the way in which real users would solve a puzzle. (For details on the puzzle solver implementation, see Appendix A.)

Each experiment, consisting of a complete puzzle solution, took roughly 10 minutes to complete (Figure 1.39). Experiments were conducted with all combinations of five variables¹, using 2-4 points along each dimension, as shown in Table 6.1. Each experiment was repeated 4 times, for a total of 384 puzzle solutions.

The Producer maintained the model, which consisted of piece descriptions and their relative positions in the completed puzzle. When pieces snapped together, the two pieces were joined into one in the model. Piece locations on the table, the highlighting of pieces under cursors, and the stacking order of overlapped pieces were represented by Perspectives.

The centralized architecture was implemented by simply disabling perspective migration. For the replicated architecture, each client was told that it was the master for each perspective, so that it could modify it locally. To synchronize peers, all such modifications were then sent to the central server², where the arrival times determined the official ordering of the modifications. These modifications were then broadcast

¹That is, excepting the prediction dimension, which only made sense with the migrating architecture.

²Peer synchronization in a replicated architecture can be implemented by peer-to-peer communication or through a central server. In our case, it was convenient to do the latter. What makes this a replicated architecture is the fact that interactions are performed locally and independently, and are synchronized later.

Dimension	Values	Comments
Architecture	Centralized, Migrating, Replicated	
Network Delay	0 ms, 50 ms, 100 ms	Each direction
Predictive Migration	no, yes	Yes only for Migrating architecture
Background Traffic	no, yes	
User Count	1, 2, 4, 6	

Table 6.1: Experiment Dimensions

to all clients. If the official ordering differed from the actual ordering at a particular replica, the out-of-order local modifications made at that client were undone and the official ordering was replayed. Otherwise the duplicate modifications made by that client and received from the central server were ignored.

Non-predictive migration was implemented by having the client request migration of a piece’s perspective when it “picked up” the piece, if it was not already the master for that perspective. Predictive migration was implemented by requesting migration of a piece’s perspective if it appeared that the local cursor was moving toward that piece and the client was not already the master for that piece’s perspective. The solver logic for determining the piece toward which the cursor should move and the prediction logic for determining the piece(s) toward which the cursor was moving were segregated so that the prediction logic would work for non-automated puzzle solutions, and to maintain the uncertainty that would exist in that case.

In the following sections, Concur will be evaluated against the criteria of Section 6.1.

6.4 Entity Types and Applications

The first three criteria, listed below, will be discussed in this section.

Criterion 1: Section 1.9 states that one of the contributions of this work is a novel classification of application software components (entities) based on their migration characteristics. This taxonomy was presented in Section 3.4. Migration times must be better than 100ms where network delays allow.

Criterion 2: I will show that these entity types can be used to build a wide range of applications by building three sample applications chosen to be representative of applications in general.

Criterion 3: By means of analyzing the above sample applications, I will argue that Concur provides a reasonable and understandable programming environment based on a centralized architecture and collaboration-unaware applications. I will demonstrate that this architecture does not require model synchronization algorithms akin to those used in replicated architectures, which simplifies the programming of the infrastructure and/or applications. I will also argue that the kinds of user mental model complexities required by a replicated architecture are not present in this architecture.

The Concur infrastructure proof of concept described in Chapter 5 supported all of the entity types of Figure 3.1 except the Timer Perspective. All of the remaining entity types were used to develop three sample programs:

- The jigsaw puzzle program used for all of the experiments,
- A simple text editor, and
- A simple pixel-based whiteboard editor.

The text and pixel editors are very simple, and these will be described in detail first. Then the puzzle program, which is substantially more complex, will be described in considerably less detail. Table 6.2 gives an overview of the components of each of these three applications.

Before continuing with a description of the sample programs, one important note must be made about Criterion 3. The multi-centered centralized architecture ensures that operations on any given entity are shown to all users in the same order. However, it does not guarantee that operations performed across multiple entities are likewise shown to all users in the same order. That is, operation O1 on perspective P1 and operation O2 on perspective P2 might be received by user U1 as (O1, O2), and by user U2 as (O2, O1). This is an incidence of the architecture bleeding through to the user that could not be prevented by the Concur architecture. It is the price of gaining much

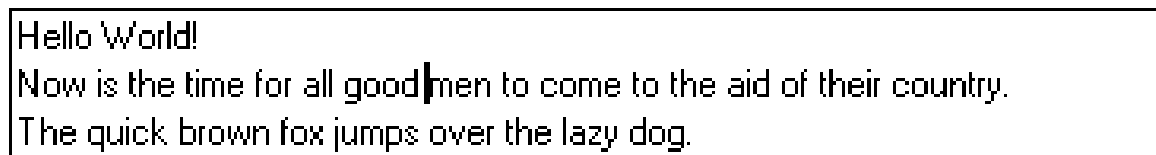
Component	Description	Text Editor	Pixel Editor	Jigsaw Puzzle
View Computation Function	Code mapping model and perspective to view specification	144	171	376
Controller Map	Code mapping events on view specification to controller	120	168	154
Controller	Controller data structure	0	0	2
Perspective Manager	Code for managing perspective(s)	0	0	248
Producer	Code managing model	33	33	96
Client Setup	Script to initiate client	22	22	89
Total		319	394	965

Table 6.2: Application Component Code Line Counts

better interactive performance. Fortunately, different orderings of operations between entities are less noticeable and less important than different orderings of operations on a single entity. In sum, Concur improves the user mental model situation dramatically, but does not eliminate it.

6.4.1 Text Editor Application

A simple text editor application is shown in Figure 6.9. In the interest of simplicity,



```
Hello World!  
Now is the time for all good|men to come to the aid of their country.  
The quick brown fox jumps over the lazy dog.
```

Figure 6.9: Text Editor Application

the editor as implemented only supports inserting characters into lines, and not, for example, deleting characters or adding or deleting lines. Completing the text editor would mostly involve the repetition of patterns in the existing example.

We'll start with the configuration of the server. Figure 6.10 shows the MetaModels.xml file, which lists the meta models, each of which defines the combination of an application with all of its inputs. There are two meta models defined for the text editor application, defining different perspectives for cursor position. If two users use the same meta model, their cursor position is shared; if they use different ones, they diverge.³ The Label attribute is used internally. The Path attribute indicates global names of the entities (across process and hosts) within Concur; it is a file-name-like path naming the meta model. The File attribute specifies the file where the meta model is defined.

³In the prototype, server configuration files are read at startup time; in a production system operations like adding perspectives would be done dynamically.


```

1 <MetaModels>
2   ...
3   <MetaModel Label="TextEditorMetaModel1"
4     Path="/User/Menges/TextEditor/Sample/MetaModel1"
5     File="TextEditorMetaModel1.xml"/>
6   <MetaModel Label="TextEditorMetaModel2"
7     Path="/User/Menges/TextEditor/Sample/MetaModel2"
8     File="TextEditorMetaModel2.xml"/>
9   ...
10 </MetaModels>

```

Figure 6.10: MetaModels.xml

The meta model definitions are shown in Figures 6.11 and 6.12. They combine

```

1 <MetaModel>
2   <ViewFunction
3     Path="/User/Menges/TextEditor/ViewFunction">
4     <Model Path="/User/Menges/TextEditor/Sample/Model"/>
5     <Cursor Path="/User/Menges/TextEditor/Sample/Cursor1"/>
6     <ControllerMap Path="/User/Menges/TextEditor/ControllerMap"/>
7   </ViewFunction>
8 </MetaModel>

```

Figure 6.11: TextEditorMetaModel1.xml

the view function with its two inputs (Model and Cursor) and its associated controller map. As you can see, the only difference is the cursor perspective name. The controller map is the code that usually maps events on the view specification into events or changes to a controller data structure being monitored by the producer (the manager of the model). In the case of this simple editor, no controller data structure is necessary, because the only operations on the model are changes to the text lines, which are performed directly by the controller map. That is, the model is a mobile model. The individual lines can migrate to the client using them, and the controller map there can operate on the master replica directly. (This greatly improves the performance of the text editor.) In this case, the producer has no work to do other

```

1 <MetaModel>
2   <ViewFunction
3     Path="/User/Menges/TextEditor/ViewFunction">
4     <Model Path="/User/Menges/TextEditor/Sample/Model"/>
5     <Cursor Path="/User/Menges/TextEditor/Sample/Cursor2"/>
6     <ControllerMap Path="/User/Menges/TextEditor/ControllerMap"/>
7   </ViewFunction>
8 </MetaModel>

```

Figure 6.12: TextEditorMetaModel2.xml

than presenting the initial model. A more complete text editor would need a controller data structure (listed in the meta model file) for invoking operations on pieces of the model larger than a line, e.g., adding and deleting lines, performing global replacements, and saving the file. The jigsaw puzzle application will demonstrate the use of a simple controller data structure for invoking application operations.

The view function, controller map, and perspective are listed in the ViewFunctions.xml, ControllerMaps.xml, and Perspectives.xml files, as shown in Figures 6.13, 6.14, and 6.15. The server needs no more model configuration, since the model is

```

1 <ViewFunctions>
2   ...
3   <ViewFunction Path="/User/Menges/TextEditor/ViewFunction"
4     File="TextEditorViewFunction.xml"/>
5   ...
6 </ViewFunctions>

```

Figure 6.13: ViewFunctions.xml

defined by the producer. We'll hold off on describing the code in the view function and controller map for a bit. The cursor perspective is defined in TextEditorCursor.xml (Figure 6.16). It doesn't actually define a cursor position yet; that will happen when a user first clicks on the editor. The same cursor definition is used for both cursors. Note that this divergence possibility is presented in the server's configuration; the application itself knows nothing of the divergence scenarios the user(s) may choose

```

1 <ControllerMaps>
2   ...
3   <ControllerMap Label="TextEditorControllerMap"
4     Path="/User/Menges/TextEditor/ControllerMap"
5     File="TextEditorControllerMap.xml"/>
6   ...
7 </ControllerMaps>

```

Figure 6.14: ControllerMaps.xml

```

1 <Perspectives>
2   ...
3   <Perspective Label="TextEditorSampleCursor1"
4     Path="/User/Menges/TextEditor/Sample/Cursor1"
5     File="TextEditorCursor.xml">
6   </Perspective>
7   <Perspective Label="TextEditorSampleCursor2"
8     Path="/User/Menges/TextEditor/Sample/Cursor2"
9     File="TextEditorCursor.xml">
10  </Perspective>
11   ...
12 </Perspectives>
13
14

```

Figure 6.15: Perspectives.xml

```

1 <Cursor/>

```

Figure 6.16: TextEditorCursor.xml

to use.

The last piece of server configuration for this application is the entry in the Applications.xml file (Figure 6.17), which specifies file name of the producer script to be launched. This script, TextEditorProducer.tcl, is shown in Figure 6.18. Line 1

```
1 <Applications>
2   ...
3   <Application Label="TextEditorProducer"
4     File="../ConcurClients/TextEditorProducer.tcl"/>
5   ...
6 </Applications>
```

Figure 6.17: Applications.xml

specifies that the Concur library should be used. Line 3 begins the definition of the TextEditorProducer class, and Line 4 specifies that this class inherits from the ConcurClient class in the Concur library. Lines 6-10 specify commands that should be imported from the library. Line 12 defines an instance-specific variable that will reference the model. Lines 14-19 define the constructor for the TextEditorProducer class. Line 14 shows that the path name of the model and the server host name and port number are passed to the constructor as arguments; this is done when an instance of the class is created on Lines 27-28. On Line 15, the server connection arguments are passed to the super-class, which creates the connection to the server. Line 17 reads the initial model from an XML file and creates a Concur tree structure representing the model. The model definition file is shown in Figure 6.19. Line 18 tells the infrastructure to replicate the model to the server. Line 23 is a Tcl construct for initializing class (static) variables. Lines 28-29 iconify the producer's windows (since the producer performs background work and does not need a UI).

The client application script is shown in Figure 6.20. It simply creates a ConcurViewer object, passing it the host name and port number and the meta model

```

1 package require ConcurLibrary
2
3 itcl::class TextEditorProducer {
4     inherit ::ConcurLibrary::ConcurClient
5
6     protected common ImportedNamespaces {
7         ConcurLibrary
8         ConcurLibrary::ConcurProcess
9         ConcurLibrary::ConcurXML
10    }
11
12    protected variable Model
13
14    public constructor {modelName ServerHost ServerPort} {
15        ConcurClient::constructor $ServerHost $ServerPort
16    } {
17        set Model [XMLFileToTree $modelName TextEditorModel.xml]
18        $Model ReplicateToChannel $ServerChannel
19    }
20
21 }
22
23 ::ConcurSafeLibrary::InitClasses
24
25 TextEditorProducer ::#auto /User/Menges/TextEditor/Sample/Model \
26     localhost 20050
27
28 wm iconify .
29 wm iconify .debug

```

Figure 6.18: TextEditorProducer.tcl

```

1 <Text>
2     <Line CDATA="Text">
3         <![CDATA[Hello World!]]>
4     </Line>
5     <Line CDATA="Text">
6         <![CDATA[Now is the time for all good men to come to the aid of their country.]]>
7     </Line>
8     <Line CDATA="Text">
9         <![CDATA[The quick brown fox jumps over the lazy dog.]]>
10    </Line>
11 </Text>

```

Figure 6.19: TextEditorModel.xml

```

1 package require ConcurLibrary
2
3 ::ConcurSafeLibrary::InitClasses
4 namespace import ::ConcurLibrary::*
5
6 set Viewer \
7     [ConcurViewer ::\#auto $Host 20050 \
8         /User/Menges/TextEditor/Sample/MetaModel1]
9
10 wm deiconify .
11 wm iconify .debug
12 wm geometry . +0+0

```

Figure 6.20: TextEditorConsumer1.tcl

path. The ConcurViewer will connect to the server, fetch the meta model, and fetch the view function, instantiating it in a secure, limited container. If one wishes to share the document but to have an independent cursor, the TextEditorConsumer2.tcl script would be used, which would reference TextEditorMetaModel2 (Figure 6.12) instead of TextEditorMetaModel1 (Figure 6.11) on Line 8.

The view function for the text editor will now be shown in pieces⁴, given its length. In this and the following discussions I will avoid boilerplate constructs already seen. First, I will show an overview, in Figure 6.21.

The first thing to note is that the code is defined within the context of an XML file. The XML file, code and all, will be loaded into a Concur tree data structure. This allows us to conveniently make certain data (below the code in the figure) available to the view function code, and to distribute the code and associated data using Concur's tree replication facilities. On Line 2, the XML attributes name the class that will define the view function, so that it can be instantiated by the infrastructure. Line 2 also specifies that the CDATA section following (which contains the code) should be

⁴Line numbers are arbitrary and may not correspond between figures.

```

1 <ViewFunction>
2   <Code ClassName="TextViewFunction" CDATA="Code">
3   <![CDATA[
4 package require ConcurSafeLibrary
5
6 itcl::class TextViewFunction {
7   # Class Initialization
8   inherit ::ConcurSafeLibrary::ConcurViewFunction
9
10  # Object Initialization
11  protected variable ControllerMap
12  protected variable Cursor
13  protected variable Model
14  protected variable ViewFunction
15
16  public constructor {pMetaModel pViewSpec pBindSpec} {
17    ::ConcurSafeLibrary::ConcurViewFunction::constructor \
18      $pMetaModel $pViewSpec $pBindSpec
19  } {
20    ...
21  }
22
23  # Protected methods
24  ...
25 }
26
27 ::ConcurSafeLibrary::InitClasses
28 ]]>
29 </Code>
30 <ViewSpecification Label="ViewSpecification" ProjectionType="Tk">
31   <Group Label="Resources"/>
32   <toplevel Label=".">
33     <frame Label="frame" borderwidth="1" background="black"/>
34   </toplevel>
35 </ViewSpecification>
36 </ViewFunction>

```

Figure 6.21: TextViewFunction.xml Overview

stored in an attribute named **Code** in the Code node of the tree. (The CDATA section in an XML file is used to store arbitrary text, so that no attempt is made to interpret it as XML.)

Line 4 indicates that the view function inherits from the *safe* Concur library, not the full Concur library. The full library is inaccessible to code running in a restricted container; the safe library only contains code that is safe to run in the container. Lines 6-8 begin the definition of the view function class, which inherits from the ConcurViewFunction class in the safe library.

The constructor on Lines 16-21 is passed the meta model, the view specification, and the bind specification tree data structures, which it then passes on to its superclass. By the time this constructor is called, the meta model has been requested from the server and instantiated locally⁵. The view and bind specifications are empty trees monitored by the infrastructure⁶. The view specification will have a projection function applied to it, and the bind specification will specify low-level event bindings that will cause events to be delivered to the controller map.

Below the code, starting at Line 30, an initial view specification data structure is defined, which will be copied by the view function to its view specification parameter. On Line 30 the projection type is specified as **Tk**. This determines the user interface technology to be used.⁷ Line 31 creates a top-level window named “.”⁸. Within that window, it creates a frame (Line 33). The user interface components corresponding

⁵Note that this would allow the meta model to change dynamically, e.g., to change from a shared to a private perspective. This type of dynamic changing of view function inputs was not used in any of the sample programs.

⁶In a full implementation of Concur, these arguments would specify only an empty portion of a larger tree, since any given view function would only create a portion of a user interface.

⁷Concur currently only supports Tk, but one could in principle define a user interface using multiple UI technologies. I did this using HTML and Tk in an earlier prototype.

⁸Actually, Tk creates this window by default, and this line just specifies that the existing one should be used. But other top-level windows can be created similarly.

to lines in the document will be created within the frame, by the view function. Note that the attributes on Line 33 (borderwidth and background) are defined by the specific UI technology being used (Tk), and are mapped to that UI by the projection function.

The TextEditorViewFunction constructor is shown in full in Figure 6.22. Lines 23-

```

19     public constructor {pMetaModel pViewSpec pBindSpec} {
20         ::ConcurSafeLibrary::ConcurViewFunction::constructor \
21             $pMetaModel $pViewSpec $pBindSpec
22     } {
23         set ViewFunctionPath [$MetaModel Get /MetaModel/ViewFunction Path]
24         set ViewFunction [* FindTree $ViewFunctionPath]
25         foreach DocumentName {ViewFunction MetaModel} {
26             set Document [set $DocumentName]
27             I$Document Interest
28             * RegisterEvent [list InterestComplete [$Document GetPath] 0 +] \
29                 [CodeClass $this ${DocumentName}InterestComplete]
30         }
31     }

```

Figure 6.22: TextEditorViewFunction Constructor

24 find the path name of the view function in the meta model and fetch a reference to the view function tree data structure. (The * command is a way of referencing commands, in this case FindTree, implemented in the infrastructure, which have been explicitly made available to code in the safe container by the infrastructure. These commands trap to the privileged infrastructure much like system calls trap to the operating system.) Line 27 requests that the ViewFunction and MetaModel tree data structures be fully replicated to the client. These data structures already exist, since the infrastructure needed them in order to instantiate the view function, but the local replica may not be complete. Recall from Section 3.5 that even though a slave replica has been obtained by a client, the entire master data structure may not be replicated locally. The tree's Interest method tells the infrastructure what portion(s) of the tree are required by the client. As invoked here, without arguments, the request is for the

entire tree. Lines 28-29 register a callback to be executed when the entire tree has been cached locally. We will now look at each of these callback methods.

The ViewFunctionInterestComplete method (Figure 6.23) is called when the entire view function has been reproduced locally. (Only the code sub-tree need have been

```

136     protected method ViewFunctionInterestComplete {} {
137         $ViewFunction Copy /ViewFunction/ViewSpecification $ViewSpec / \
138         -tags -recurse
139     }

```

Figure 6.23: TextEditorViewFunction ViewFunctionInterestComplete

requested by the infrastructure.) Lines 137-138 copy the initial view specification from the bottom of Figure 6.21 to the ViewSpec tree monitored by the projection function in the infrastructure. The projection function will respond by creating the frame in the default top-level window (.).

The MetaModelInterestComplete method (Figure 6.24) requests replicas of all the other trees specified in the meta model of Figure 6.11 (that is, the Model, the Cursor, and the ControllerMap), by traversing the /MetaModel/ViewFunction sub-tree and calling the RequestParameter command on each of its descendant nodes. The

```

62     protected method MetaModelInterestComplete {} {
63         I$MetaModel IApply /MetaModel/ViewFunction \
64         -precommand [CodeClass $this RequestParameter]
65     }

```

Figure 6.24: TextEditorViewFunction MetaModelInterestComplete

RequestParameter method (Figure 6.25) makes the appropriate request and registers the method ParameterReceived to be called when each replica has been received.

The ParameterReceived method (Figure 6.26) tells the infrastructure that the view function is interested in the entire Model, Cursor, and ControllerMap trees,

```

110     protected method RequestParameter {I} {
111         switch -- [$MetaModel Get $I Type] {
112             Cursor - Model - ControllerMap {
113                 set Path [$MetaModel Get $I Path]
114                 * RequestReplica $Path
115                 * RegisterEvent [list ReplicaReceived $Path] \
116                     [CodeClass $this ParameterReceived $I]
117             }
118         }
119     }

```

Figure 6.25: TextEditorViewFunction RequestParameter

and registers corresponding InterestComplete methods to be called when these are available.

```

100     protected method ParameterReceived {I} {
101         set Label [$MetaModel Label $I]
102         set $Label [* FindTree [$MetaModel Get $I Path]]
103         set Parameter [set $Label]
104         I$Parameter Interest
105         * RegisterEvent \
106             [list InterestComplete [$Parameter GetPath] 0 +] \
107             [CodeClass $this ${Label}InterestComplete $I]
108     }

```

Figure 6.26: TextEditorViewFunction ParameterReceived

Until this point, the methods of the view function are pretty much boilerplate, and could be abstracted into the super-class and eliminated from the sub-class. It is instructive, however, to discuss them, as a means of understanding Concur, and the methods presented give full control to the view function so that it can efficiently tell the infrastructure exactly what it does and does not want to see. From this point on, the methods are mostly specific to the text editor application.

When the model replica is fully received, the ModelInterestComplete method (Figure 6.27) sets up callbacks for whenever any nodes in the model are created, moved,

or deleted. (The current sample program is really only interested in the creation of

```
67     protected method ModelInterestComplete {I} {
68         foreach Operation {create move delete} {
69             I$Model INotify create -$Operation \
70                 [CodeClass $this ModelNotify_$Operation]
71         }
72         if {[I$Model Exists /Text]} {
73             foreach Child [I$Model Children /Text] {
74                 CreateLine $Child
75             }
76         }
77     }
```

Figure 6.27: TextEditorViewFunction ModelInterestComplete

nodes, but the other operations would be needed for a complete text editor implementation.) Then, if the /Text node exists⁹ (see Figure 6.19), the CreateLine method is called for each of its children (one per line in the document). The ModelNotify_create callback (Figure 6.28) also calls CreateLine when a new line is created. The Create-

```
87     protected method ModelNotify_create {Op I args} {
88         if {[I$Model Get $I Type] == {Line}} {
89             CreateLine $I
90         }
91     }
```

Figure 6.28: TextEditorViewFunction ModelNotify_create

Line method (Figure 6.29) sets up a callback for whenever that line’s text is changed, and creates a Tk “entry” UI component in the view specification’s frame. The entry component displays a single line of text. When the text of a line in the model

⁹Why wouldn’t the /Text node exist? The InterestComplete method is called when the *server* has finished sending whatever it has of the document to the client; it may not have received the entire document from the producer yet.

```

44     protected method CreateLine {I} {
45         I$Model ITraceAttribute $I Text [CodeClass $this ModelTextChanged]
46         $ViewSpec Insert /ViewSpecification/./frame \
47             -Label entry[$Model Position $I] \
48             -data [list Type entry borderwidth 0 Text [$Model Get $I Text] \
49                 width 60 _expand true _fill both]
50     }

```

Figure 6.29: TextEditorViewFunction CreateLine

changes, the ModelTextChanged callback (Figure 6.30) updates the appropriate node in the view specification, and the projection function reacts by changing the text in the entry component.

```

82     protected method ModelTextChanged \
83     {Tree I Key Ops PrevValue Value SetCommand} {
84         set Line [$Model Position $I]
85         $ViewSpec Set /ViewSpecification/./frame/entry$Line Text \
86             $Value $SetCommand
87     }

```

Figure 6.30: TextEditorViewFunction ModelTextChanged

The CursorInterestComplete method registers a callback to be executed when the Position attribute of the cursor changes¹⁰. If the Position attribute already exists (recall that it did not yet exist in Figure 6.16), it calls the SetCursor method (Figure 6.31). The CursorPositionChanged callback method (Figure 6.32) also calls

```

124     protected method SetCursor {Value} {
125         foreach {Line Character} $Value {}
126         $ViewSpec Set /ViewSpecification/./frame/entry$Line Cursor $Character
127     }

```

Figure 6.31: TextEditorViewFunction SetCursor

¹⁰Using tree event bubbling, it should be possible to register a callback only on the /Text node of the model. Unfortunately, I have not yet implemented bubbling for attribute tracing in the infrastructure.

SetCursor whenever the position attribute changes. The SetCursor method sets the

```
59     protected method CursorPositionChanged \
60     {Tree I Key Ops PrevValue Value SetCommand} {
61         SetCursor $Value
62     }
```

Figure 6.32: TextEditorViewFunction CursorPositionChanged

current cursor position for the line containing the cursor. See the text insertion character before the word “men” in Figure 6.9. (Only the active line’s cursor will be displayed and used by the application.)

The ControllerMapInterestComplete method (Figure 6.33) instantiates the controller map in the container in which the view specification is running, passing along the meta model, controller map, view specification, and bind specification. This

```
36     protected method ControllerMapInterestComplete {I} {
37         namespace eval :: [$ControllerMap Get /ControllerMap/Code Code]
38         namespace eval :: \
39             [list [$ControllerMap Get /ControllerMap/Code ClassName] \
40                 ::\#auto $MetaModel $ControllerMap $ViewSpec \
41                     $BindSpec]
42     }
```

Figure 6.33: TextEditorViewFunction ControllerMapInterestComplete

method could also be considered boilerplate.

That’s it - the entire view function is 144 lines of code (including data represented in XML), and could be considerably shortened by moving boilerplate code to the super-class. Note that the code is not aware of how many users are sharing any of its inputs, and that it does not have to be involved in any synchronization of peer replicas. It is written for a centralized architecture. Note also that the view function is implemented as a mathematical *function*, i.e., it reacts to its inputs by changing its outputs, in a manner such that the same inputs always produce the

same outputs. Since the view function is implemented in an imperative language, it need not implement a function, but it does. More will be said about this issue in Chapter 7.

Now we will turn to the `ControllerMap` code, and here again, we will look at it piece by piece, because of its length. We begin with the overview of Figure 6.34. The main thing to note from this figure is the `BindSpecification` node at the bottom. This is where the controller map would typically define its low-level event bindings on the view specification. In this case, the bindings need to be applied to each Tk entry UI component, and these aren't created until the model replica appears, so the `BindSpecification` node is empty¹¹.

Figure 6.35 shows the controller map's constructor. On Line 22, it copies the empty bind specification to the infrastructure-monitored `BindSpec`. Since the bind specification is empty in this case, this is a no-op. Then it obtains references to the cursor and model trees (Lines 26-34). Finally, it sets up callbacks on the view specification, so that it knows when entry components are created there by the view function, and it can set up event bindings on them. (First it registers a callback so that it will know when the entry components are added (Line 36), and then it sets up bindings for any entry components that may already exist (Lines 37-39)). The `ViewSpecNotify_create` callback method is shown in Figure 6.36.

The `CreateBindings` method is shown in Figure 6.37. This method creates event bindings on newly-created entry elements in the view specification. There are two event bindings - one for keystrokes and the other for Button-1 button presses. The keystrokes insert characters into a line, and the button press changes the location of

¹¹A better approach using tags is implemented for the pixel editor, and is described in the next section. That approach could not be used here because tags are not yet implemented in the Concur infrastructure for most Tk UI elements, such as entry components.

```

1 <ControllerMap>
2   <Code ClassName="TextEditorControllerMap" CDATA="Code">
3   <![CDATA[
4 package require ConcurSafeLibrary
5
6 itcl::class TextEditorControllerMap {
7   # Class Initialization
8   inherit ::ConcurSafeLibrary::ConcurControllerMap
9
10  # Object Initialization
11
12  protected variable Cursor
13  protected variable Model
14
15  public constructor {
16    pMetaModel pControllerMap pViewSpec pBindSpec
17  } {
18    ::ConcurSafeLibrary::ConcurControllerMap::constructor \
19      $pMetaModel $pControllerMap $pViewSpec $pBindSpec
20  } {
21    ...
22  }
23
24  # Protected Methods
25  ...
26 }
27
28 ::ConcurSafeLibrary::InitClasses
29 ]]>
30   </Code>
31   <BindSpecification Label="BindSpecification"/>
32 </ControllerMap>

```

Figure 6.34: TextEditorControllerMap.xml Overview


```

16 public constructor {
17     pMetaModel pControllerMap pViewSpec pBindSpec
18 } {
19     ::ConcurSafeLibrary::ConcurControllerMap::constructor \
20     $pMetaModel $pControllerMap $pViewSpec $pBindSpec
21 } {
22     $ControllerMap Copy /ControllerMap/BindSpecification $BindSpec / \
23     -recurse -tags
24
25
26     set Cursor \
27     [* FindTree \
28     [$MetaModel Get /MetaModel/ViewFunction/Cursor \
29     Path]]
30
31     set Model \
32     [* FindTree \
33     [$MetaModel Get /MetaModel/ViewFunction/Model \
34     Path]]
35
36     I$ViewSpec INotify create -create [CodeClass $this ViewSpecNotify_create]
37     foreach Child [$ViewSpec Children /ViewSpecification/./frame] {
38         CreateBindings $Child
39     }
40 }

```

Figure 6.35: TextEditorControllerMap Constructor

```

90 protected method ViewSpecNotify_create {Op I} {
91     CreateBindings $I
92 }

```

Figure 6.36: TextEditorControllerMap ViewSpecNotify_create

```

44     protected method CreateBindings {I} {
45         set Position [$ViewSpec Position $I]
46         $BindSpec Insert /BindSpecification -label Key$Position \
47             -data [list \
48                 Type Binding \
49                 Sequence "<Key>" \
50                 ViewSpecPath /ViewSpecification/./frame/entry$Position \
51                 ParameterString "K %K A %A" \
52                 Callback [CodeClass $this Key]]
53         $BindSpec Insert /BindSpecification -label Button-1$Position \
54             -data [list \
55                 Type Binding \
56                 Sequence "<Button-1>" \
57                 ViewSpecPath /ViewSpecification/./frame/entry$Position \
58                 Callback [CodeClass $this Button-1]]
59         I$ViewSpec ITraceAttribute $I Text \
60             [CodeClass $this ViewSpecTextChanged]
61     }

```

Figure 6.37: TextEditorControllerMap.xml CreateBindings

the insertion cursor. The Sequence attribute specifies which Tk event should generate callbacks. The ViewSpecPath attribute defines the node in the view specification corresponding to the entry to which the binding should apply. The optional ParameterString attribute tells the infrastructure what event information should be passed along to the callback - in the case of the Key binding these are K (the key cap name of the key pressed, e.g., “Return”) and A (the key value itself, e.g., the newline character or the letter “a”). In addition, the CreateBindings method sets up a callback when the text on a line in the view specification changes. This is needed to adjust the cursor rightward when characters are added before the cursor on the line, either by the local user or a remote one.

The Key callback method is shown in Figure 6.38. This method is passed a reference to the (entry) node in the view specification corresponding to the event (I), the parameter string containing event information described in the last paragraph, the position of the insertion character, and the position of the mouse (the latter

```

63     protected method Key {I ParameterString Character Mouse} {
64         array set Parameter $ParameterString
65         if {$Parameter(A) == {}} {return}
66
67         set ViewSpecLine [$ViewSpec Position $I]
68         set ModellLine [$Model NthChild /Text $ViewSpecLine]
69         switch -- $Parameter(K) {
70             Return {
71                 }
72             default {
73                 $Model ApplyToMaster $ModellLine {} \
74                 [list [$Model Get / id] InsertAttribute \
75                     $ModellLine Text $Parameter(A) $Character]
76             }
77         }
78     }

```

Figure 6.38: TextEditorControllerMap Key

not being needed for keystrokes). Line 64 assigns the elements in the parameter string to a hash table. Line 65 ignores all keystroke events except those that actually insert a character. (For example, it ignores shift key presses.) Lines 67-68 determine which document line in the model should be affected. Lines 73-75 effect the insertion of the character into the appropriate line of the model. The extra complexity in this call (ApplyToMaster) enables the infrastructure to find the master copy of the line and insert the character in that copy, thus supporting migration of lines. The InsertAttribute call inserts characters into an attribute at a specified point. This is better than retrieving the attribute value, changing the value by inserting the new character, and re-setting the value, because it enables the infrastructure to know exactly what change was made. This allows for efficiency enhancements (e.g., inserting the character into the entry component using that UI component's operation, rather than setting the entire value of the component). It also enables certain required functionality, such as re-positioning the cursor only if characters inserted are to the

left of the cursor.

Updating the cursor in this case is handled by the ViewSpecTextChanged method (Figure 6.39). I'll leave understanding of the details of this method to the reader,

```
94     protected method ViewSpecTextChanged \
95     {Tree I Key Ops PrevValue Value SetCommand} {
96         set ViewSpecLine [$ViewSpec Position $I]
97         set CommandName [lindex $SetCommand 0]
98         if {$CommandName == {InsertAttribute}} {
99             foreach {CommandName Index Attribute String Position} \
100                 $SetCommand {}
101             set CursorPosition [$Cursor Get /Cursor Position]
102             foreach {CursorLine CursorCharacter} $CursorPosition {}
103             if {[string length $String] > 0
104                 && $ViewSpecLine == $CursorLine
105                 && $Position <= $CursorCharacter} {
106                 set NewPosition [list $ViewSpecLine \
107                     [expr {$CursorCharacter+[string length $String]}]]
108                 $Cursor ApplyToMaster /Cursor {} [list [$Cursor Get / id] \
109                     Set /Cursor Position $NewPosition]
110             }
111         }
112     }
```

Figure 6.39: TextEditorControllerMap ViewSpecTextChanged

since it doesn't have a lot to do with Concur. A similar algorithm could be used to handle the deletion of characters or the insertion of lines before the cursor.

Figure 6.40 shows the Button-1 callback method. This method changes the cursor

```
80     protected method Button-1 {I ParameterString Insert Character} {
81         set ViewSpecLine [$ViewSpec Position $I]
82         $Cursor ApplyToMaster /Cursor {} \
83             [list [$Cursor Get / id] Set /Cursor Position \
84                 [list $ViewSpecLine $Character]]
85         $Cursor FetchMasterToken [$Cursor Index /Cursor]
86         set ModellLine [$Model NthChild /Text $ViewSpecLine]
87         $Model FetchMasterToken $ModellLine
88     }
```

Figure 6.40: TextEditorControllerMap Button-1

position in response to the mouse button press. It then requests master status for

both the cursor perspective and the line of the mobile model corresponding to the line the user clicked. If it receives master status for these nodes (meaning that they have migrated to the local host), future updates will be faster for that user while he is editing the line.

This concludes our discussion of the text editor application. The entire application consists of about 319 lines of collaboration-unaware code, written to the centralized architecture. The user mental model is that of the simpler centralized architecture, meaning that synchronization issues are not exposed to the user.

6.4.2 Pixel Editor Application

The pixel editor application (Figure 6.41) is very similar in structure to the text editor application. The main difference is in the way that units of the mobile model are defined. Rather than migrating lines, as in the text editor, the pixel editor migrates square areas (tiles) of the whiteboard, laid out in a grid. The boundaries of these 50-pixel square areas are shown in Figure 6.42. This presents some challenges not seen in the text editor application. This section will focus on the code that is unique to the pixel editor.

As was the case with the text editor sample program, the pixel editor sample program is very simple, in order to make its implementation easy to understand. The pixel editor is only able to draw pixels (not objects like lines or rectangles), and only in one color. No eraser capability is implemented.

The meta models (Figure 6.43) are similar to the text editor's, except that the cursor perspective has been replaced by a Color perspective. This perspective would allow the pixel editor to use different color "pens", and users could either share a pen color or use different ones. Since this would be similar to how cursors are implemented

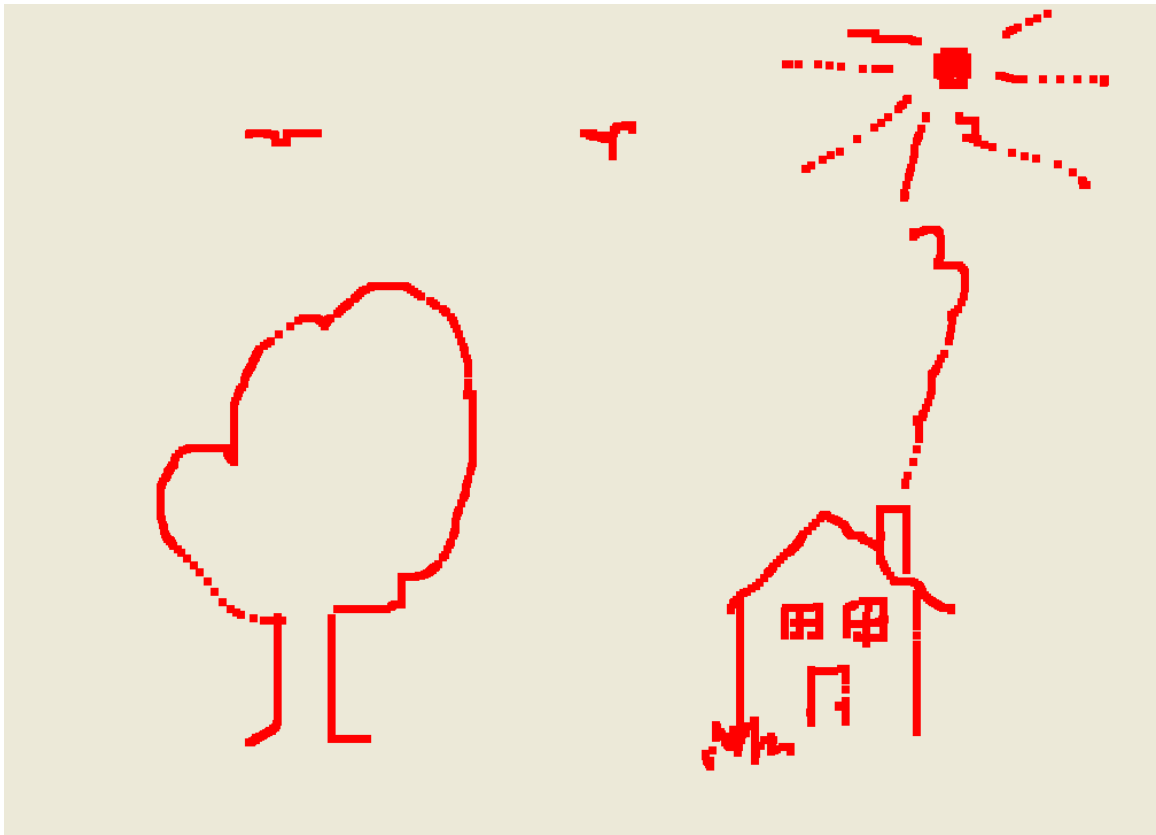


Figure 6.41: Pixel Editor Application

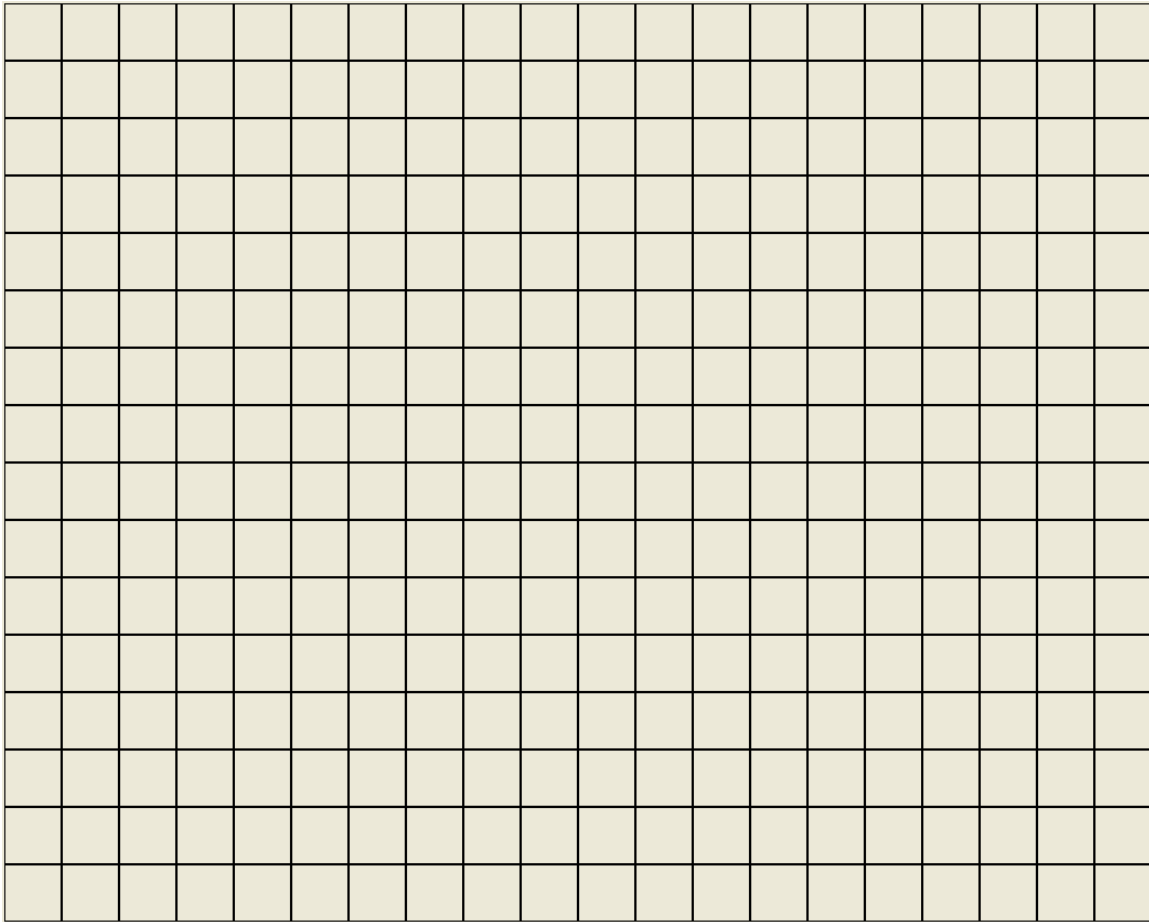


Figure 6.42: Pixel Editor Application Showing Area Boundaries

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <MetaModel>
3   <ViewFunction
4     Path="/User/Menges/PixelEditor/ViewFunction">
5     <Model Path="/User/Menges/PixelEditor/Sample/Model"/>
6     <Color Path="/User/Menges/PixelEditor/Sample/Color1"/>
7     <ControllerMap Path="/User/Menges/PixelEditor/ControllerMap"/>
8   </ViewFunction>
9 </MetaModel>
```

Figure 6.43: Pixel Editor MetaModel1

in the text editor, this functionality has not actually been implemented in the pixel editor. The model (Figure 6.44) initially displays empty squares in each of the 50-pixel tiles.

The View Function (Figure 6.45) is similar to the text editor’s, except that the data portion at the bottom is different. On Line 39, there is a Group node called “Resources”. The Tk API separates the notion of an image resource (which contains the actual pixels) from an image drawn into a window (which references the resource). The image resources will be placed under the “Resources” group node. A group node is just a grouping mechanism for nodes; nothing interesting there. Instead of a frame, a canvas resides under the top-level (Line 33). A Tk canvas UI component is essentially a drawing editor that allows one to place lines, rectangles, ovals, images, other UI components (e.g., entry components), etc. arbitrarily onto a rectangular canvas¹². The most interesting difference from the text editor is the “Tags” group and the “Image” tag. The group was created here to group all tags, but there’s only one, so it’s not required. The “Image” tag provides a place to apply bindings that will automatically apply to *all* the images on the canvas. We’ll see how that’s done when we look at the controller map.

When tiles show up in the model (Figures 6.46 and 6.47), the UpdateViewSpec method (Figure 6.48) is called. On Lines 46-47, this method sets up a callback for when the image data for a tile changes. Then it creates the image resource corresponding to the tile, and finally it creates the tile component on the canvas. Note that this tile component has a tag of **Image**, so that any event bindings for the tag of that name will apply to this component. In this way, all image components on

¹²On Line 34, the width, height and scrollregion are hardwired to a given size; one should either implement scrolling or make this dynamically dependent on the model size. Hardwiring them here is just a shortcut for this example.


```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <Whiteboard>
3   <Row>
4     <Tile CDATA="data" format="png" width="50" height="50"><![CDATA[ivBORw0KGgcAAAANSUHEUgAAADIAAAAYCAQAAAHDMXCsAA
5 REFUWIXtwQENAAAawQD3T20ON6AAAAAGHMDc+cAaale0+FAAAAAUUVORK5CYII=]]></Tile>
6     ...
7   </Row>
8   <Row>
9     ...
10  </Row>
11 </Whiteboard>

```

Figure 6.44: Pixel Editor Model

```

6 itcl::class PixelEditorViewFunction {
7     # Class Initialization
8     inherit ::ConcurSafeLibrary::ConcurViewFunction
9
10    # Object Initialization
11    protected variable ControllerMap
12    protected variable Color
13    protected variable Model
14    protected variable ViewFunction
15
16    public constructor {pMetaModel pViewSpec pBindSpec} {
17        ::ConcurSafeLibrary::ConcurViewFunction::constructor \
18            $pMetaModel $pViewSpec $pBindSpec
19    } {
20        ...
21    }
22
23    # Protected methods
24    ...
25 }
26
27 ::ConcurSafeLibrary::InitClasses
28 ]]>
29 </Code>
30 <ViewSpecification Label="ViewSpecification" ProjectionType="Tk">
31     <Group Label="Resources"/>
32     <toplevel Label=".">
33         <canvas Label="canvas" borderwidth="0" _expand="true" _fill="both"
34             width="1000" height="800" scrollregion="0 0 999 799">
35             <Group Label="Tags">
36                 <Tag Label="Image" Tag="Image" anchor="nw"/>
37             </Group>
38         </canvas>
39     </toplevel>
40 </ViewSpecification>
41 </ViewFunction>

```

Figure 6.45: PixelEditorViewFunction.xml Overview

```

91     protected method ModelInterestComplete {I} {
92         I$Model IApply / -precommand [CodeClass $this UpdateViewSpec]
93         foreach Operation {create move delete} {
94             I$Model INotify create -$Operation \
95                 [CodeClass $this ModelNotify_$Operation]
96         }
97     }

```

Figure 6.46: PixelEditorViewFunction ModelInterestComplete

```

105     protected method ModelNotify_create {Op I args} {
106         UpdateViewSpec $I
107     }

```

Figure 6.47: PixelEditorViewFunction ModelNotify_create

```

44     protected method UpdateViewSpec {I} {
45         if {[{$Model Get $I Type] == {Tile}}} {
46             I$Model ITraceAttribute $I data \
47                 [CodeClass $this ModelTileChanged]
48             set Row [$Model Position $I/..]
49             set Column [$Model Position $I]
50             $ViewSpec Insert /ViewSpecification/Resources \
51                 -label imageR${Row}C$Column \
52                 -data [list \
53                     Type image \
54                     ImageType photo \
55                     data [$Model Get $I data] \
56                     format [$Model Get $I format] \
57                     width [$Model Get $I width] \
58                     height [$Model Get $I height] \
59                 ]
60             set Height [$Model Get $I height]
61             set Width [$Model Get $I width]
62             $ViewSpec Insert /ViewSpecification/./canvas \
63                 -label tileR${Row}C$Column \
64                 -data [list \
65                     Type image \
66                     image imageR${Row}C$Column \
67                     Coords [list [expr {$Column*$Width}] \
68                             [expr {$Row*$Height}]] \
69                     tags [list Image] \
70                 ]
71         }
72     }

```

Figure 6.48: PixelEditorViewFunction UpdateViewSpec

the canvas will trigger the same bindings. The pixel editor controller map therefore does not need to create any bindings dynamically as images are created in the view specification.

When the model image data for a tile changes, the `ModelTileChanged` method (Figure 6.49) is called. This method simply updates the image resource with the

```

99     protected method ModelTileChanged {Tree I Key Ops PrevValue Value SetCommand} {
100         set Row [$Model Position $I/..]
101         set Column [$Model Position $I]
102         $ViewSpec Set /ViewSpecification/Resources/imageR${Row}C$Column data $Value $SetCommand
103     }

```

Figure 6.49: PixelEditorViewFunction ModelTileChanged

changes to the model tile image. The projection function will pick up this change and effect the changes to the image on the display. The pixel editor does not need to monitor the view specification in the way that the text editor did to update cursor positions, since there is no insertion cursor.

The pixel editor controller map overview is shown in Figure 6.50. Bindings are set up at the bottom for the mouse Button-1 click and for dragging the mouse with button 1 depressed. The `ParameterString` specifies that the x and y coordinates of the mouse should be passed to the callback routine when the event is triggered. Note again that the bindings are set up using the `Image` tag, so they apply to all images on the canvas.

The `ButtonPress-1` callback routine (Figure 6.51) simply calls the `Draw` routine to draw a dot at the position of the mouse, and requests master status for the tile under the cursor. Once it receives master status (i.e., the tile has migrated), drawing within that tile area will be a local interaction, and thus will be much faster for this user. Note that the `Row` and `Column` of the tile should be obtainable from `Tk`, but `Tk` does not always generate the event on the right tile. So this code computes the row and

```

6 itcl::class PixelEditorControllerMap {
7     # Class Initialization
8     inherit ::ConcurSafeLibrary::ConcurControllerMap
9
10    # Object Initialization
11    protected variable Color
12    protected variable Model
13
14    public constructor {
15        pMetaModel pControllerMap pViewSpec pBindSpec
16    } {
17        ::ConcurSafeLibrary::ConcurControllerMap::constructor \
18            $pMetaModel $pControllerMap $pViewSpec $pBindSpec
19    } {
20        ...
21    }
22
23    # Protected Methods
24    ...
25 }
26
27 ::ConcurSafeLibrary::InitClasses
28 ]]>
29 </Code>
30 <BindSpecification Label="BindSpecification">
31     <Binding Label="ButtonPress-1" Sequence="&lt;ButtonPress-1&gt;"
32         ParameterString="x %x y %y"
33         ViewSpecPath="/ViewSpecification/./canvas/Tags/Image"/>
34     <Binding Label="B1-Motion" Sequence="&lt;B1-Motion&gt;"
35         ParameterString="x %x y %y"
36         ViewSpecPath="/ViewSpecification/./canvas/Tags/Image"/>
37 </BindSpecification>
38 </ControllerMap>

```

Figure 6.50: PixelEditorControllerMap Overview

```

57     protected method ButtonPress-1 {I ParameterString} {
58         set Label [$ViewSpec Label $I]
59         array set P $ParameterString
60
61         # Calculate these myself because Tk appears to be inaccurate
62         # with respect to which tile was clicked.
63         set Row [expr {int($P(y)/50)}]
64         set Column [expr {int($P(x)/50)}]
65
66         set Tile [$Model NthChild [$Model NthChild /Whiteboard $Row] $Column]
67         Draw $Tile $Row $Column $P(x) $P(y)
68         $Model FetchMasterToken $Tile
69     }

```

Figure 6.51: PixelEditorControllerMap ButtonPress-1

column number based on the x and y position of the mouse. It currently hard-codes the width (50) and height (50) of tiles; these should be determined dynamically from the model instead. The code for the B1-Motion event is identical to the code for the ButtonPress-1 event, thus the code can be shared. This means that migration is requested on every mouse movement, so that if the user draws into a new tile, the new tile will also migrate. (If the tile is already migrated, requesting migration is an efficient, local operation.)

The Draw routine is complicated by the fact that the pen width is not one pixel. This means that all drawing calls must check to see if they are near the edge of one tile, so that they can appropriately draw into adjacent tiles as necessary. Due to its size, I have not included a figure for the Draw routine.

This concludes our discussion of the pixel editor application. It is about the same complexity as the text editor application, but it does call out different issues with respect to mobile model definition.

6.4.3 Jigsaw Puzzle Application

The jigsaw puzzle is a much more complete application than the text editor or the pixel editor. It was used for all of the performance experiments described in this chapter. I will describe this application at a higher level of abstraction, simply showing method signatures and code fragments, and describing the functionality they implement.

The puzzle meta model (Figure 6.52) defines a view function, a model, a perspective, and a controller map. The perspective in this case is compound; it contains one

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <MetaModel>
3   <ViewFunction
4     Path="/User/Menges/Puzzle/ViewFunction">
5     <Model Path="/User/Menges/Puzzle/Family/Model"/>
6     <Perspective Label="FragmentPerspective"
7       Path="/User/Menges/Puzzle/Family/FragmentPerspective1"/>
8     <ControllerMap Path="/User/Menges/Puzzle/ControllerMap"/>
9   </ViewFunction>
10 </MetaModel>
```

Figure 6.52: PuzzleMetaModel1.xml

independently-migratable node for each piece of the puzzle. Each node defines several attributes of that piece, which all migrate together. the perspective is also more complex in that producer-like code is required to monitor the set of perspectives and manage snap operations (since these involve multiple pieces). Since providing for this type of code was not part of the original Concur design, more investigation will be required to determine if the solution implemented in the current prototype is the best solution. Essentially, my solution was to associate perspective manager code with the perspective. This code is instantiated by the server automatically, and run in a secure container there. In this way, the perspective remains independent of the application,

and multiple perspective instances can still be created to implement divergence.

An overview of the Puzzle producer is shown in Figure 6.53. The interesting thing

```
1 package require ConcurLibrary
2
3 itcl::class PuzzleProducer {
4     inherit ::ConcurLibrary::ConcurClient
5
6     ...
7
8     public constructor {modelName ControllerName ServerHost ServerPort} {
9         ConcurClient::constructor $ServerHost $ServerPort
10    } {
11        ...
12        $Controller RegisterTreeEvent Snap [CodeClass $this Snap]
13        ...
14    }
15
16    # Protected Methods
17    protected method Snap {Index1 Index2} {
18        ..
19    }
20 }
21
22 ::ConcurSafeLibrary::InitClasses
23
24 PuzzleProducer ::#auto /User/Menges/Puzzle/Family/Model \
25     /User/Menges/Puzzle/Family/Controller localhost 20050
26
27 wm iconify .
28 wm iconify .debug
```

Figure 6.53: PuzzleProducer.tcl Overview

to note about this producer is that it registers an interest in the **Snap** event on the controller. This event is applied to the controller by the perspective manager when a user attempts to snap pieces together and the perspective manager determines that such an operation is appropriate (i.e., the pieces are close enough together, and they match). The snap method effects the snap operation by joining two puzzle fragments into one in the model.

The model itself is shown in Figure 6.54¹³. Note that the puzzle is composed of

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <Puzzle height="287" width="400" CanvasHeight="700" CanvasWidth="1000">
3   <ImageData>
4     <Image Label="R0C0" Width="58" Height="60" CDATA="data" format="png">
5       <![CDATA[
6 iVBORw0KGgoAAAANSUhEUgAAADoAAAA8CAYAAAF5ZmJAAAABGdBTUEAAAYagMeiWXwAAF3xJ
7       ...
8       ]]>
9     </Image>
10    ...
11  </ImageData>
12  <Frag>
13    <Frag Label="R0C0" Coords="-2 -2" Pieces="1" AdjFrag="R0C1 R1C0">
14      <Piece Label="R0C0" Coords="0 0" Color="#7f8f4e"
15        NWColor="121" NEColor="231" SWColor="342" SEColor="551">
16        <Edges>
17          <Edge Label="Top" Direction="Straight"
18            Coords="2 2 42 2"/>
19          <Edge Label="Right" Direction="Right" AdjPiece="R0C1"
20            Coords="42 2 41 22 54 15 54 30 41 21 41 42"/>
21          <Edge Label="Bottom" Direction="Down" AdjPiece="R1C0"
22            Coords="41 42 21 42 27 57 16 55 22 42 2 41"/>
23          <Edge Label="Left" Direction="Straight"
24            Coords="2 41 2 2"/>
25        </Edges>
26      </Piece>
27    </Frag>
28    ...
29  </Frag>
30 </Puzzle>

```

Figure 6.54: Puzzle Model

fragments, where a fragment is a set of connected puzzle pieces. A piece is composed of edges Top, Right, Bottom, and Left. Each edge defines its coordinates and the direction of the protruding key or intruding lock (if any). Both fragments and edges identify adjacent fragments and edges, which facilitates the determination of whether or not a piece can snap together with a nearby piece. Pieces also call out their average color and the average color of each of the four quadrants of the piece; these are used

¹³The model was generated by a Tcl script that takes an image and cuts it into puzzle pieces.

to allow players to favor certain colors (so that they are not all working on the same pieces), and to prioritize their match attempts (using the quadrant colors).

Figure 6.55 highlights several interesting things. Let's look at the view specification at the bottom first. The canvas contains tags for edges, images, invisible polygons, and the puzzle as a whole. This enables the code to selectively apply operations to sets of related UI components en masse. Tags are also set up dynamically for each fragment and piece. Invisible polygons are overlaid on pieces to catch events and apply them to the appropriate piece. Groups are set up for containing both the visible and invisible polygons of from which the pieces are composed. The order of pieces in these groups determines the stacking order of the pieces. A background rectangle (table top) is also defined.

Several highlights of the view function code will now be pointed out. When UI components are created in the view specification corresponding to the model and perspective, the infrastructure is told to link attributes between the model and perspective on one hand, and the UI components on the other. In this way, attributes (such as coordinate positions) are automatically updated in the view specification when they change in the model or perspective, so that the code does not have to explicitly monitor the source of these changes and make corresponding changes in the target. Finally, note that the view function monitors changes in the model (which occur when pieces are snapped together), and in the perspective (when pieces are snapped, and when reference counts change) and make the corresponding changes to the view specification. Reference counts are used to determine when a fragment should be highlighted; it is highlighted when one or more user cursors are over the piece.

Now we turn briefly to the controller map of Figure 6.56. As you can see, it is composed of event handlers for various UI events. The code for the controller map

```

1 <ViewFunction>
2   <Code ClassName="PuzzleViewFunction" CDATA="Code">
3     ...
4 itcl::class PuzzleViewFunction {
5     ...
6     protected method InitViewSpec {I} {
7         ...
8         CreateFragment $I $Label
9         ...
10        $Model LinkAttribute Coords $I $ViewSpec $Canvas/Tags/P$Label
11        ...
12    }
13
14    protected method CreateFragment {I Label} {
15        ...
16        $FragmentPerspective LinkAttribute Coords \
17            /Fragments/$Label $ViewSpec $Canvas/Tags/F$Label
18        ...
19    }
20
21    protected method ModelNotify_delete {Op I} {...}
22    protected method ModelNotify_move {Op I} {...}
23    protected method FragmentPerspectiveNotify_move {Op I} {...}
24    protected method FragmentPerspectiveRefCountChanged
25        {Tree I Key Ops PrevValue Value SetCommand}{...}
26 }
27
28 ::ConcurSafeLibrary::InitClasses
29 ]]>
30 </Code>
31 <ViewSpecification Label="ViewSpecification" ProjectionType="Tk">
32   <Group Label="Resources"/>
33   <toplevel Label=".">
34     <canvas Label="canvas" borderwidth="0" _expand="true" _fill="both">
35       <Group Label="Tags">
36         <Tag Label="Edge" Tag="Edge" capstyle="projecting" fill=""
37             smooth="true" width="3"/>
38         <Tag Label="Image" Tag="Image" anchor="nw"/>
39         <Tag Label="InvisiblePolygon" Tag="InvisiblePolygon" fill=""
40             outline="" smooth="true" width="3"/>
41         <Tag Label="Puzzle" Tag="Puzzle"/>
42       </Group>
43       <Group Label="Invisible"/>
44       <Group Label="Visible"/>
45       <rectangle Label="Background" tags="Background" Coords="0 0 0 0"
46           fill="LightGoldenrod" outline="" stipple="gray50"/>
47     </canvas>
48   </toplevel>
49 </ViewSpecification>
50 </ViewFunction>

```

Figure 6.55: PuzzleViewFunction.xml Overview

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <ControllerMap>
3   <Code ClassName="PuzzleControllerMap2" CDATA="Code">
4   <![CDATA[
5   ...
6   itcl::class PuzzleControllerMap2 {
7     ...
8     # Protected Methods
9     protected method Background_Enter {args} {...}
10    protected method Canvas_Leave {args} {...}
11    protected method Canvas_Motion {I ParameterString} {...}
12    protected method Image_Enter {args} {...}
13    protected method Item_B1-Motion {I ParameterString} {...}
14    protected method Item_ButtonPress-1 {I ParameterString} {...}
15    protected method Item_ButtonPress-3 {I ParameterString} {...}
16    protected method Item_ButtonRelease-1 {I ParameterString} {...}
17    protected method Item_Enter {I ParameterString} {...}
18    protected method Item_Snap {I ParameterString} {...}
19    ...
20  }
21
22  ::ConcurSafeLibrary::InitClasses
23 ]]>
24   </Code>
25   <BindSpecification Label="BindSpecification">
26     <Binding Label="Canvas_Leave" Sequence="&lt;Leave&gt;"
27       ViewSpecPath="/ViewSpecification/./canvas"/>
28     <Binding Label="Canvas_Motion" Sequence="&lt;Motion&gt;"
29       ParameterString="x %x y %y"
30       ViewSpecPath="/ViewSpecification/./canvas"/>
31     <Binding Label="Background_Enter" Sequence="&lt;Enter&gt;"
32       ViewSpecPath="/ViewSpecification/./canvas/Background"/>
33     <Binding Label="Image_Enter" Sequence="&lt;Enter&gt;"
34       ViewSpecPath="/ViewSpecification/./canvas/Tags/Image"/>
35     <Binding Label="Item_B1-Motion" Sequence="&lt;B1-Motion&gt;"
36       ParameterString="x %x y %y"
37       ViewSpecPath="/ViewSpecification/./canvas/Tags/InvisiblePolygon"/>
38     <Binding Label="Item_ButtonPress-1" Sequence="&lt;ButtonPress-1&gt;"
39       ParameterString="x %x y %y"
40       ViewSpecPath="/ViewSpecification/./canvas/Tags/InvisiblePolygon"/>
41     <Binding Label="Item_ButtonRelease-1" Sequence="&lt;ButtonRelease-1&gt;"
42       ParameterString="x %x y %y"
43       ViewSpecPath="/ViewSpecification/./canvas/Tags/InvisiblePolygon"/>
44     <Binding Label="Item_ButtonPress-3" Sequence="&lt;ButtonPress-3&gt;"
45       ParameterString="x %x y %y"
46       ViewSpecPath="/ViewSpecification/./canvas/Tags/InvisiblePolygon"/>
47     <Binding Label="Item_Enter" Sequence="&lt;Enter&gt;"
48       ViewSpecPath="/ViewSpecification/./canvas/Tags/InvisiblePolygon"/>
49     <Binding Label="Item_Snap" Sequence="&lt;ButtonPress-2&gt;"
50       ParameterString="x %x y %y"
51       ViewSpecPath="/ViewSpecification/./canvas/Tags/InvisiblePolygon"/>
52   </BindSpecification>
53 </ControllerMap>

```

Figure 6.56: PuzzleControllerMap.xml Overview

is about 160 lines long. Of course, the enhanced controller map that adds the puzzle solver is much longer and more complex, coming in at around 1700 lines of code.

Finally, we'll look at the perspective manager discussed above and shown in Figure 6.57. As you can see, it inherits from `ConcurSafeObject`, so that it can run in a

```
7 itcl::class PuzzleFragmentPerspectiveManager {
8     # Class Initialization
9
10    inherit ::ConcurSafeLibrary::ConcurSafeObject
11
12    ...
13
14    public constructor {pPerspectiveDef} {...}
15
16    # Protected methods
17
18    protected method InitPerspective {I} {...}
19    protected method PerspectiveCoordsModified
20    {Tree I Attribute Ops PXY XY SetCommand} {...}
21    protected method PerspectiveSnap
22    {Tree I Attribute Ops PrevValue Value SetCommand} {...}
23    protected method ModelNotify_delete {Operation I} {...}
24    ...
25 }
26 ::ConcurSafeLibrary::InitClasses
27 ]]>
28 </Code>
29 </PerspectiveManager>
```

Figure 6.57: PuzzleFragmentPerspectiveManager.xml Overview

protected container. It is responsible for creating the perspective tree, which records the coordinates, stacking order, pick count (how many users have the piece picked up), reference count (how many users have their cursors over the piece) and snap attribute (used to request a snap attempt). When a snap attempt is requested, the perspective manager determines if it is ok to snap the pieces together (i.e., a nearby

piece fits and is close enough to the correct position of the one on which the snap request was made). If so, it requests that the producer snap the pieces by applying a Snap event to the controller. The perspective manager also monitors model changes by the producer, so that it can merge perspectives when puzzle fragments are snapped together. The entire perspective manager is around 250 lines of code.

6.4.4 Application Summary

In summary, the three applications developed for Concur are reasonable in complexity, as measured by the ease of describing how they work, and the lines of code required to implement them. They are written to the centralized architecture, are collaboration unaware, and do not contain the peer synchronization logic often required in a replicated collaborative system. Future investigation should include the development of a number of larger applications with a large number of perspective types and a complete user interface, but the work that has been done is enough to give convincing arguments that programming to the Concur infrastructure is reasonable and that it avoids the complexity of replicated architectures. As a result, it also avoids complexity in the user's mental model caused by the replicated architecture.

6.5 Determinism

Criterion 4: I will argue that Concur is deterministic.

State machines are problematic with respect to determinism, especially if there are multiple state machines running in different environments, with hard-to-eliminate unknown inputs, and internal black-box state that is difficult or impossible to access. I have argued in this dissertation that using functions instead of state machines is

a valuable software engineering technique, particularly in collaborative systems. It facilitates reliable and accurate sharing, support of latecomers and mobility, and treatment of view computation as a black box. I have illustrated the ease with which functional view computation can be implemented using an imperative language, through the examples described above. Functional computations (in this case, from trees to tree or DAGs to DAG) are deterministic in that they always produce the same output for a given set of inputs.

However, ideally one should be able to *guarantee*, at the language level, that a view function is, in fact, functional (deterministic). That is, it should be possible to define a language that efficiently computes and recomputes a view function as its inputs change, and that also guarantees that the same output is always produced for a given set of inputs.

Early in this work I investigated functional languages to use for this purpose. I was not able to find a functional language that handled incremental re-evaluation of its output based on incremental changes to its inputs. That is, using existing functional languages, one would have to re-evaluate the entire function for every input change, which is not efficient enough for practical use.

During the last few months of my research for this dissertation, I re-considered this problem. I came to the conclusion that the development of such a language would not be terribly difficult, taking the following approach. Suppose one developed a functional language *along with* a full set of primitive functions implemented in an imperative language, and *certified* to be functional (i.e., their functionality not guaranteed by the imperative language, but by examination and/or proof techniques). Then, any view function coded as a composition of these primitives would in turn be functional. The design of such a language is left for future research.

6.6 Divergence and Modes of Work

Criterion 5: I will argue that above-mentioned entities can be used to implement a wider range of desirable per-participant divergence scenarios than existing systems.

Criterion 6: I will argue that Concur supports individual work as well as all four classes of collaborative work illustrated in Figure 1.1, and that Concur supports transitions among all these forms of work. In the process, I will show how Concur supports latecomers and mobility.

By this point, it should be apparent that a very large range of desirable divergence scenarios can be implemented by Concur. Chapter 4 contains a lengthy discussion of the kinds of divergence that might be implemented, and the sample programs show how a given divergence scenario might be reasonably implemented. Further investigation of larger, more complex programs would contribute to our understanding of the link between the units of state desirable for migration and those desirable for divergence, but the examples provided certainly point to a strong link between the two. Divergence is most commonly desired in the elements of state that are *not* part of the essential semantics maintained by the model. It is usually most useful when applied to small bits of state that determine how this essential state is to be viewed. These same small bits of state are the easiest to migrate, and the most likely to benefit from migration, because they typically change more quickly and frequently than the essential state represented by the model. It also seems clear that the range of divergence scenarios in Concur is much larger than in previous systems, which tended to hard-code divergence possibilities into the infrastructure.

Concur's support for functional view computation clearly facilitates the support of latecomers and mobility. In Concur, applications are not aware (or, at least, need not be aware) of how many users are participating in a conference. Thus, support for

differences between zero, one, and more than one participant, and transitions among these, are clearly transparent. This means that transitions among the four quadrants of Figure 1.1 and the special (but most common) case of individual work are also supported transparently.

6.7 Performance

Criterion 7: I will deliver a prototype infrastructure supporting these applications, built as a multi-centered centralized system with entity migration support.

This criterion was met by the Concur infrastructure implementation described in Chapter 5.

In this section I will present the results of experiments using that infrastructure, in order to demonstrate how criteria 8 and 9 were met.

Criterion 8: I will demonstrate that this infrastructure and applications conforming to it give substantially better interactive performance to all participants than does a purely centralized architecture, and close to that of a replicated architecture. I will demonstrate that this architecture scales better in terms of both processor and network bandwidth utilization than a purely centralized architecture.

This criterion addresses latency and scalability. As measured in our experiments, latency is the user-perceived delay between the time a user takes an action (such as dragging a piece) and the time he actually sees the result of that action (e.g., when the piece actually moves in his projection). Network delays of 0 ms, 50 ms, and 100 ms were introduced into the experiments to simulate local area networks (LANs), Intranets, and the Internet (Wide Area Network, or WAN), respectively. (See [AKSJ03] for a paper describing TCP round-trip times, which I cite as evidence

that my chosen introduced network delays are reasonable.) These delays are one-directional, so the actual latency introduced was 0 ms, 100 ms, and 200 ms. It is the user-perceived latency that normally distinguishes centralized (long latencies) architectures from replicated ones (short latencies), and it is the short replicated latencies that we were attempting to achieve in the migrating architecture. Latencies (e.g., when dragging a single puzzle piece across the screen) often overlap as shown in Figure 6.58, giving the user much better performance than he would have if he

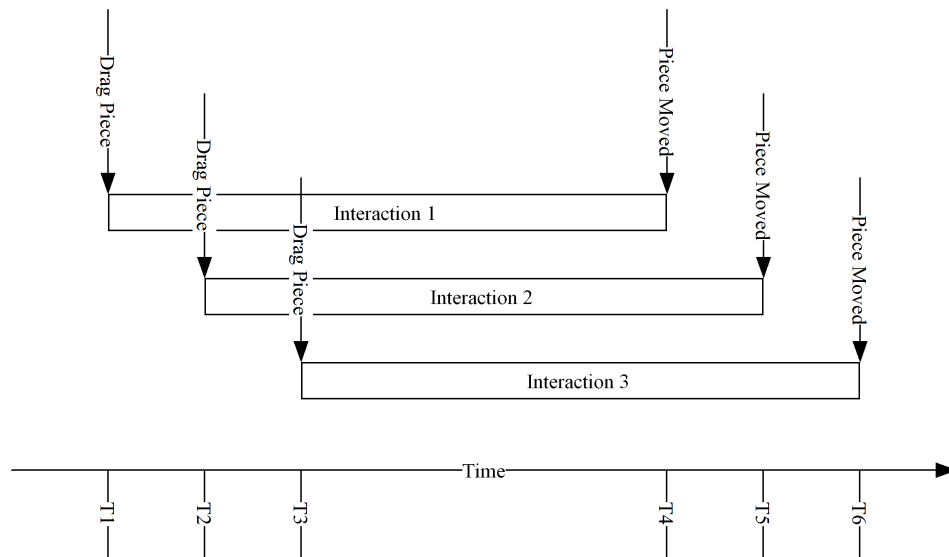


Figure 6.58: Overlapping Latencies while Dragging a Puzzle Piece

had to wait for each round trip to occur before the next one began. Our latency measurements are, for example, from T2 to T5. Figure 6.59 shows the impact of the artificially-introduced latencies on user-perceived latencies.

Figure 6.60 is arguably the most important result of this dissertation. It shows that the migrating architecture's latencies are similar to those of the replicated architecture, while, as was argued earlier in this chapter, the programming model remains that of the centralized architecture. It also shows that the migrating and replicated

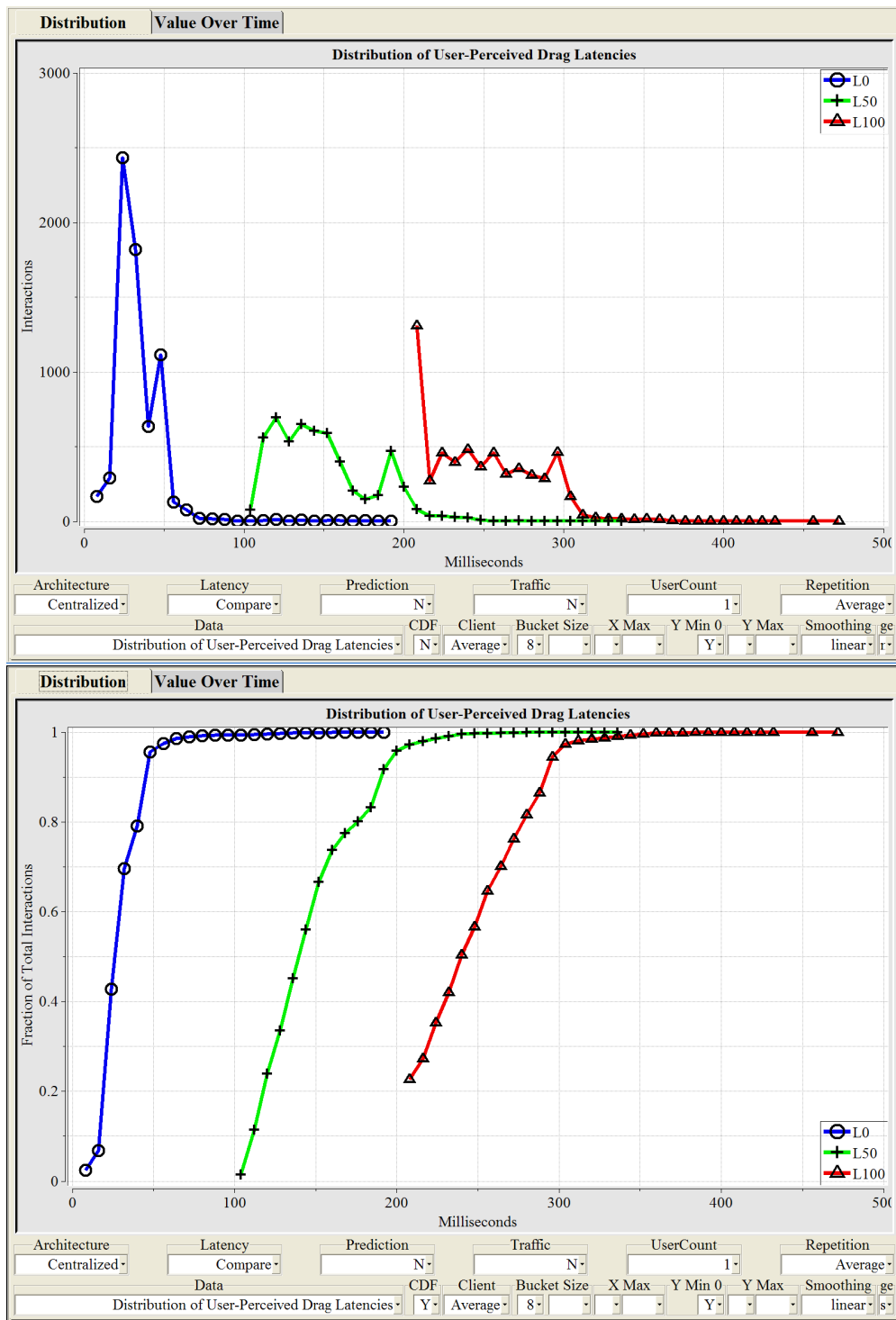


Figure 6.59: User Perceived Latency Distribution by Introduced Latencies

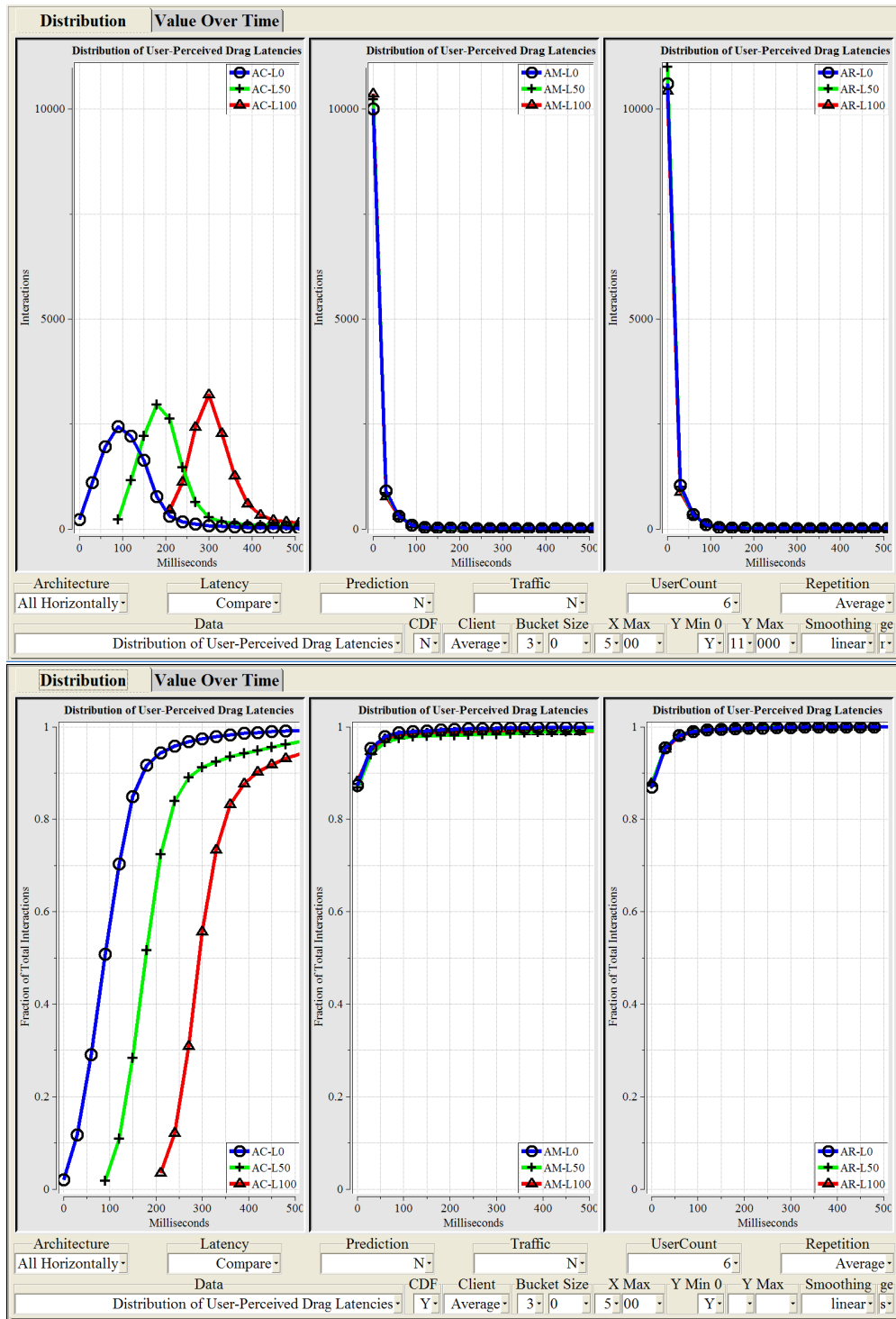


Figure 6.60: User Perceived Latency Distributions by Architecture

architecture latencies are not affected significantly by latencies introduced by the network.¹⁴ Figure 6.61 zooms in to show just how close the migrating and replicated latencies are.

The next two figures show latency distributions as they vary by background traffic (Figure 6.62) and user count (Figure 6.63). There is an almost imperceptible impact on latencies in the centralized architecture caused by background traffic. Note also the impact of user count on latencies; significant for the centralized architecture, but minor for the migrating and replicated architectures. Figure 6.64 zooms in to show the impact of user count on latency for these latter two architectures in more detail.

Figure 6.63 serves to demonstrate one important scalability result: that the migrating architecture scales as well as the replicated architecture, and much better than the centralized architecture, in terms of user-perceived latency as the number of users grows. In other respects (e.g., resource utilization), scalability results were less dramatic. These were discussed in Section 1.10.3.

Task completion times, shown in Figure 1.39 were a little shorter for the migrating architecture than for the centralized architecture, but a little longer than the replicated architecture. This topic was also discussed in Section 1.10.3. One might expect the differences between the architectures to be more pronounced for longer experiments, and perhaps with better tuning of the migrating architecture's migration algorithms, e.g., to avoid migration thrashing. See Chapter 7 for a discussion of this latter point.

In Section 6.2 I noted that the client machines used in these experiments were severely under-powered by today's standards. I will now briefly discuss the costs (in CPU time) of the units of work performed by the client initiating an action, the

¹⁴One of my readers asked why the tails of the graphs in Figure 6.60 are so long (500+ ms). An analysis of these long latencies is given in Appendix B.

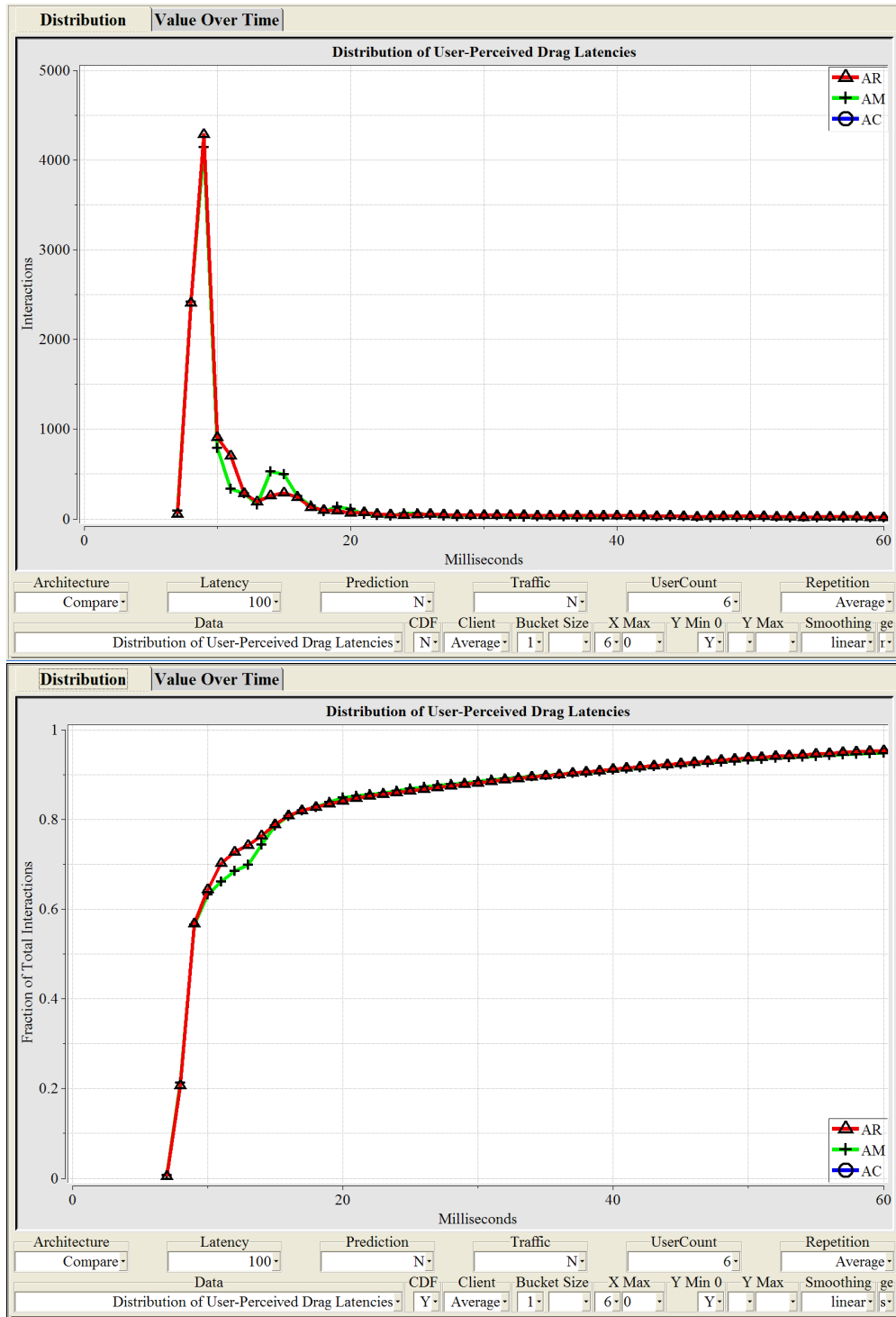


Figure 6.61: Migrating and Replicated Latencies (The Centralized plot is off the graph to the right.)

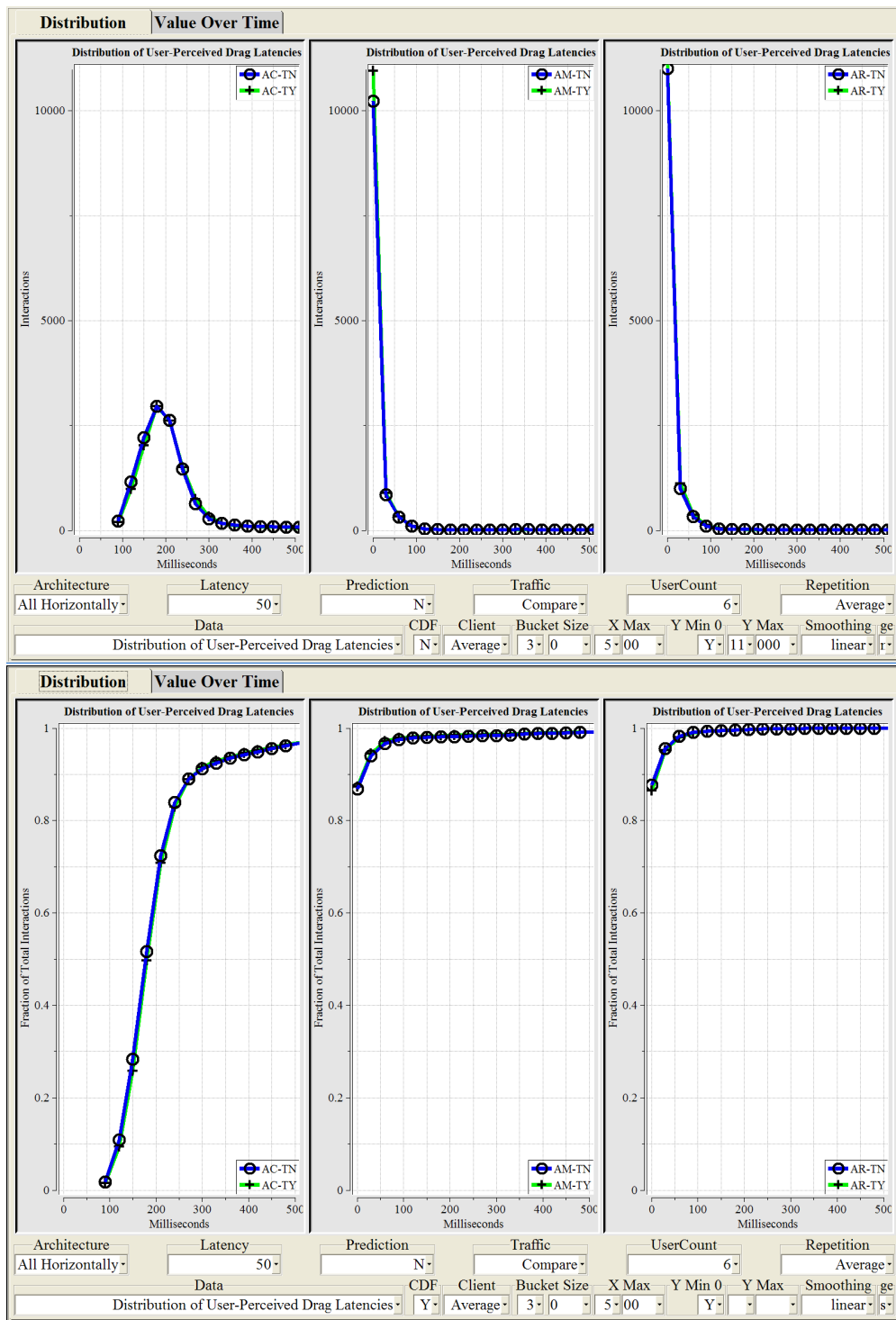


Figure 6.62: Latency Distribution by Background Traffic

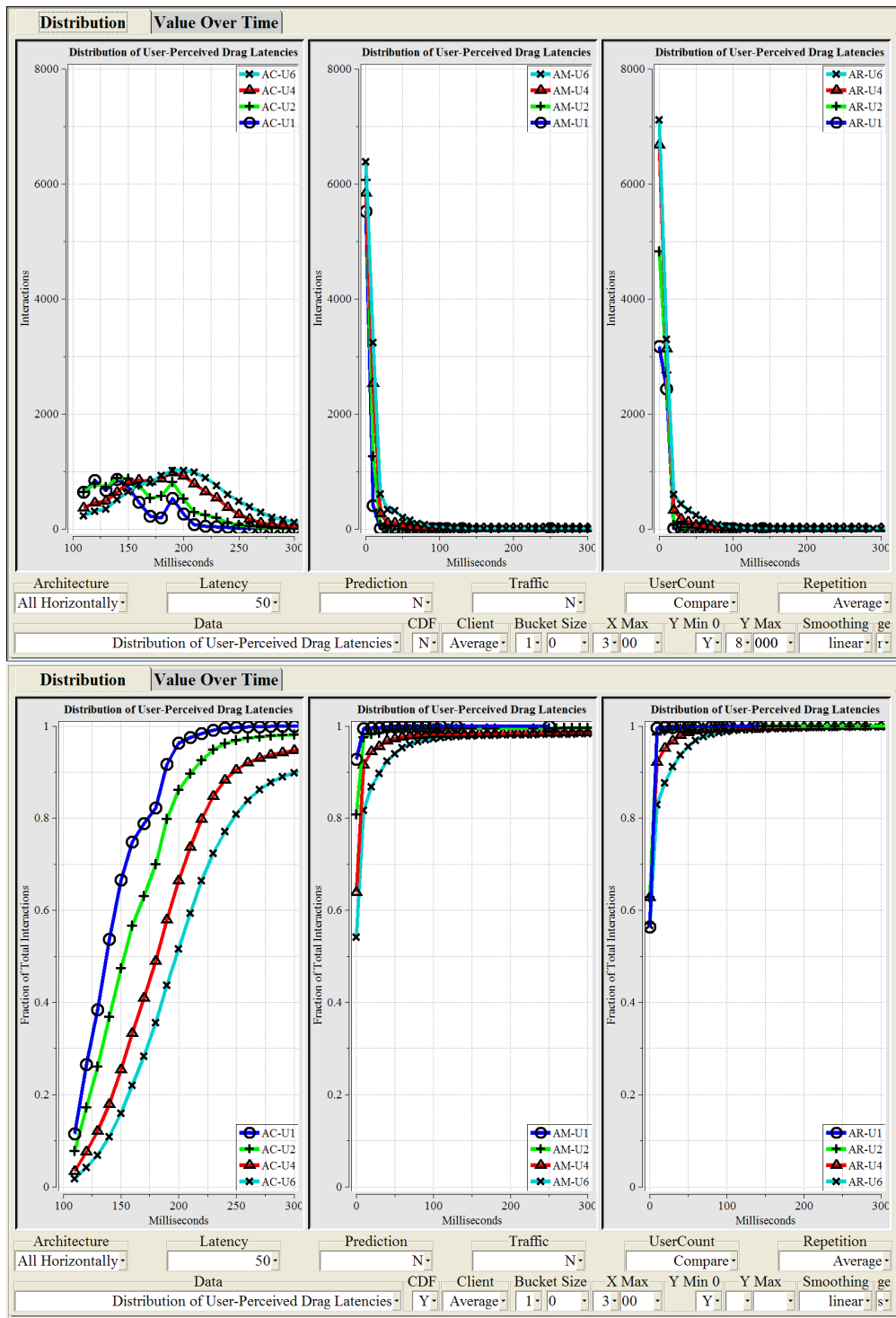


Figure 6.63: Latency Distribution by User Count

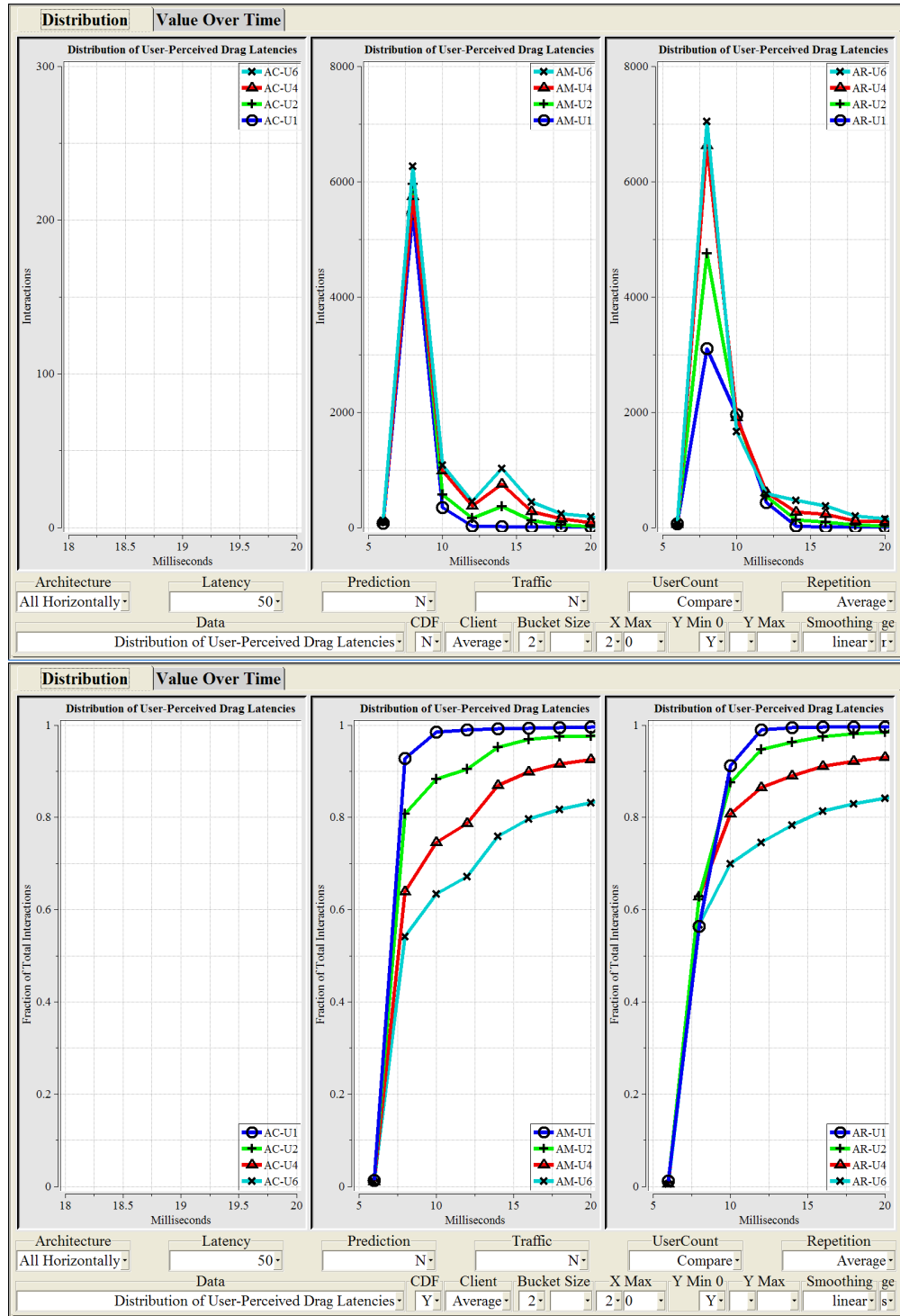


Figure 6.64: Latency Distribution by User Count, Migrating and Replicated (The Centralized plot is off the graph to the right.)

other clients, and server, and how these costs are affected by the migration algorithms employed. This analysis will give the reader an idea of how these algorithms might perform on the machines of today and of the future.

Figure 6.65 shows that the migrating architecture costs about 3-4 ms more on the initiating client when a piece is picked up, plus an occasional additional cost of about 6 ms when the piece actually migrates. This is the cost of checking, requesting, and receiving the migration of pieces to the local computer. The other clients show an occasional cost in the low tens of ms (Figure 6.66) when a piece migrates away from them. Figure 6.67 shows an occasional server cost of several ms when a piece migrates.

Figure 6.68 demonstrates at, on the initiating client, the migrating architecture costs about 2 ms less than the other architectures for each mouse movement, when dragging a piece. This is attributable to the fact that the client does not need to receive a message describing mouse movements it makes when a piece is migrated locally. (As I have implemented the replicated architecture, it does need to receive such a message in order to perform peer synchronization.) The gain here outweighs the loss in piece pickup times, because piece movements are much more frequent than piece pickups. The other clients and the server are not significantly affected by the architecture in this scenario, so their graphs are not shown.

Figure 6.69 shows an occasional cost in the tens and low hundreds of ms moving the cursor using the cursor vectoring prediction algorithm. This cost is incurred when a piece actually migrates. Figure 6.70 Shows that a corresponding cost is only incurred on other clients when this prediction algorithm is enabled and when migration actually occurs. Figure 6.71 shows a corresponding, but much smaller, cost on the server.

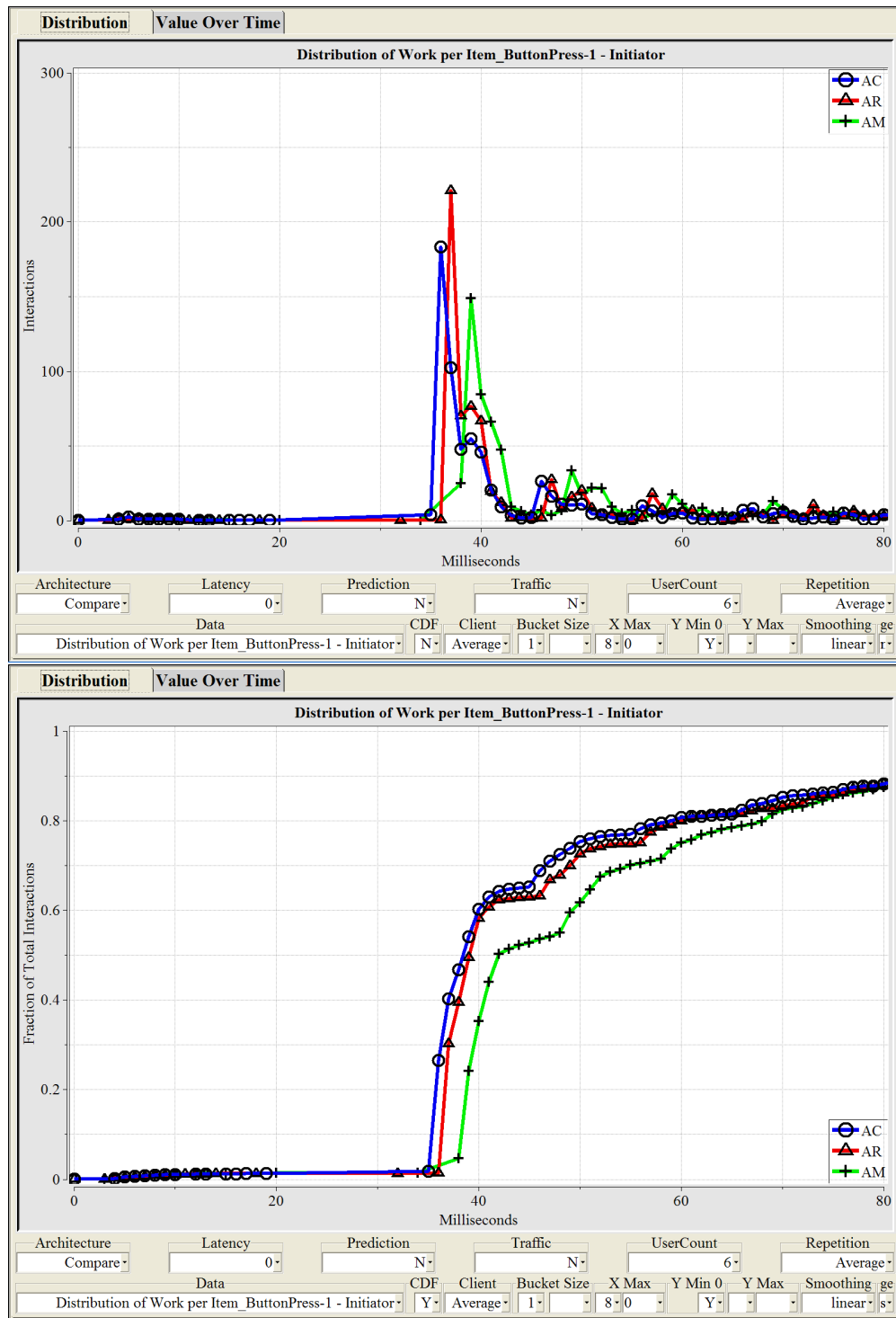


Figure 6.65: Distribution of Work Picking Up Piece by Architecture - Initiator

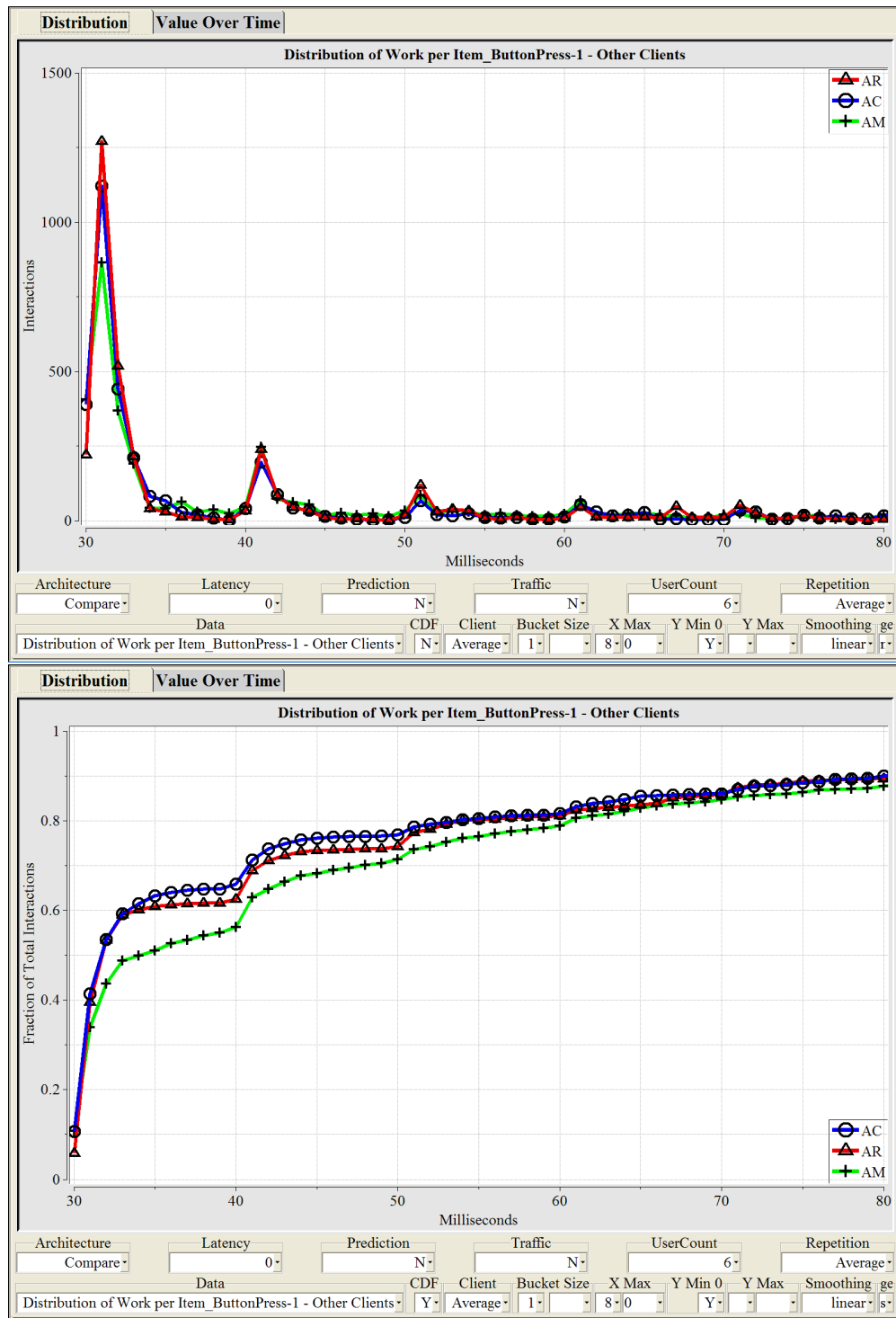


Figure 6.66: Distribution of Work Picking Up Piece by Architecture - Other Clients

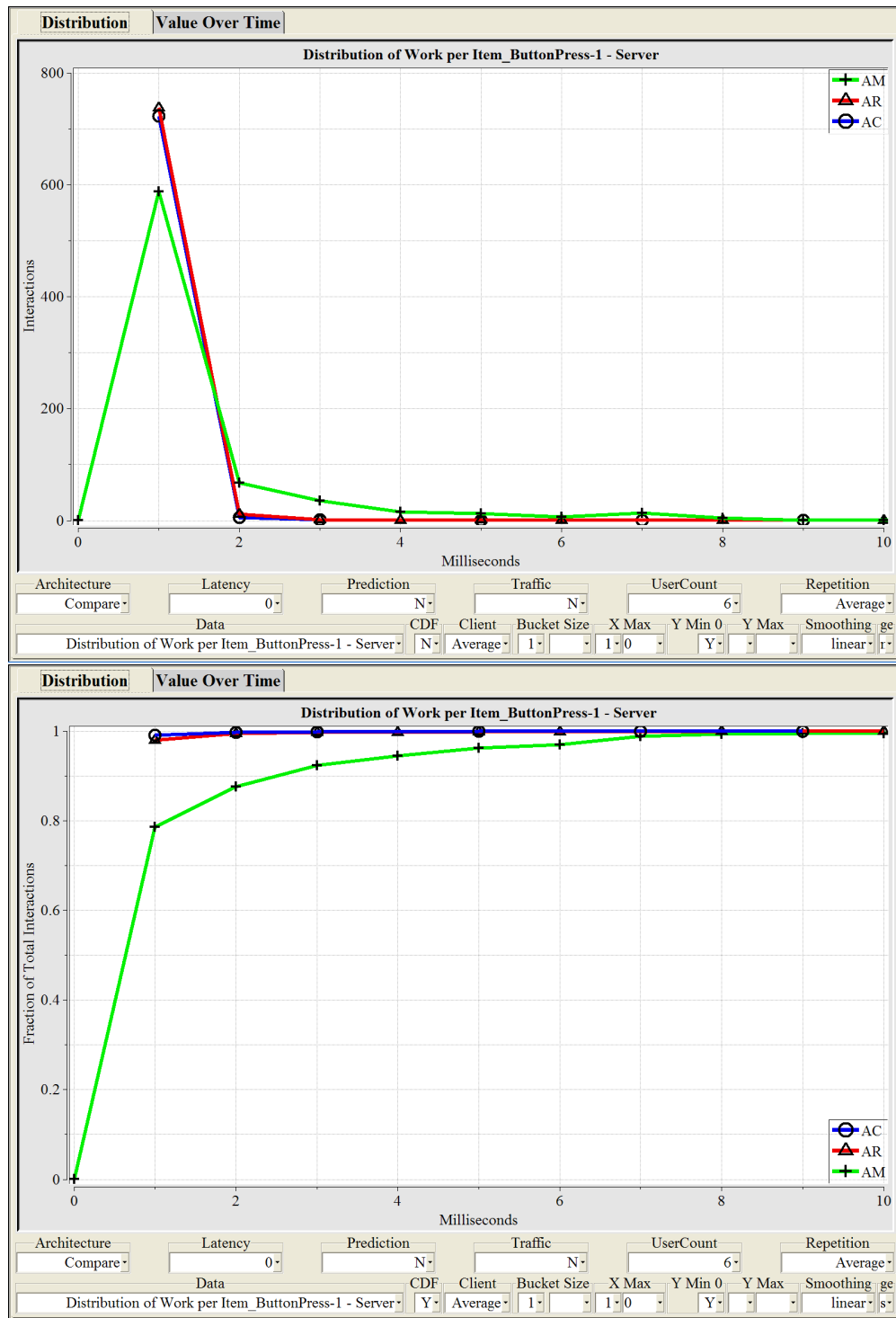


Figure 6.67: Distribution of Work Picking Up Piece by Architecture - Server

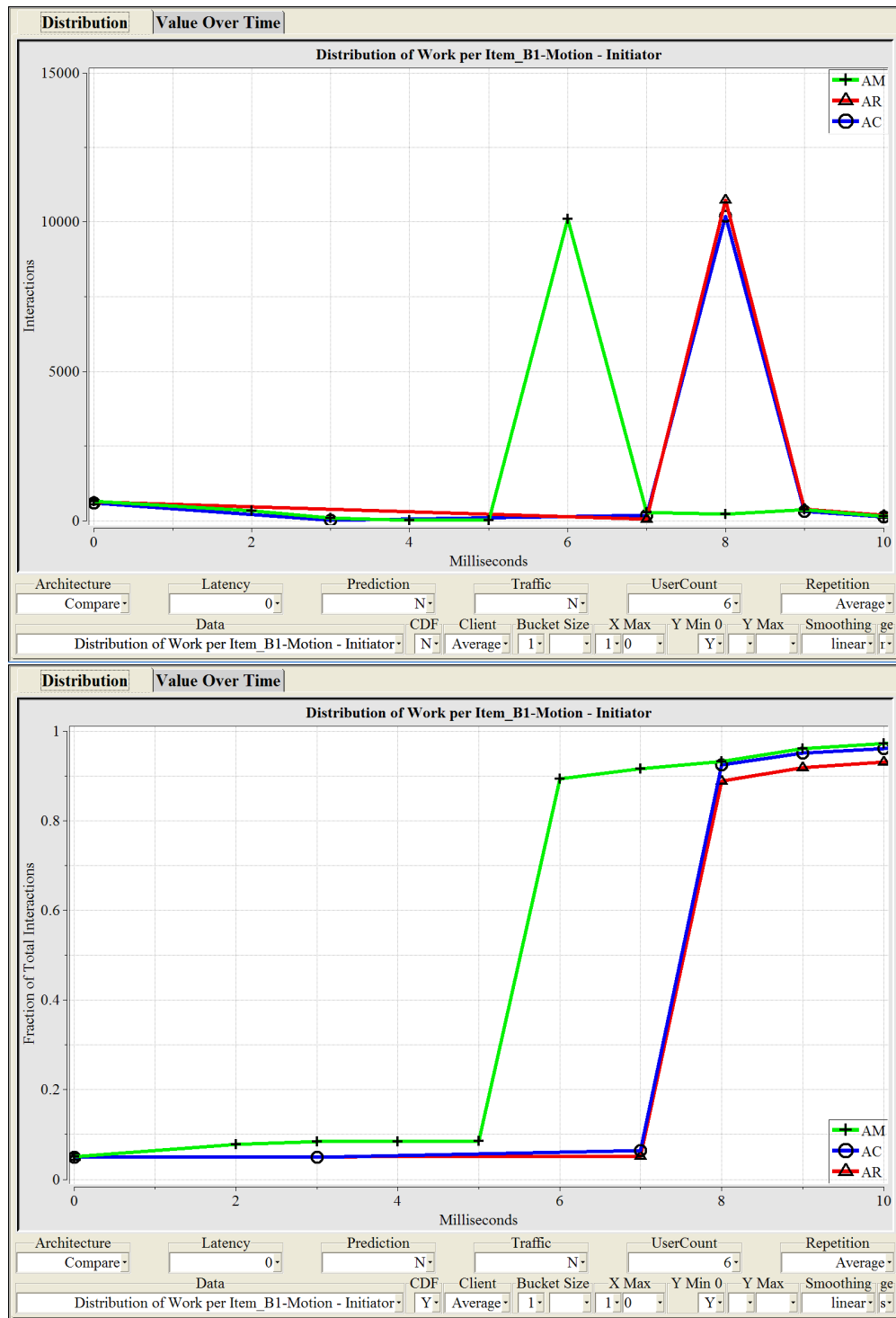


Figure 6.68: Distribution of Work Dragging a Piece by Architecture

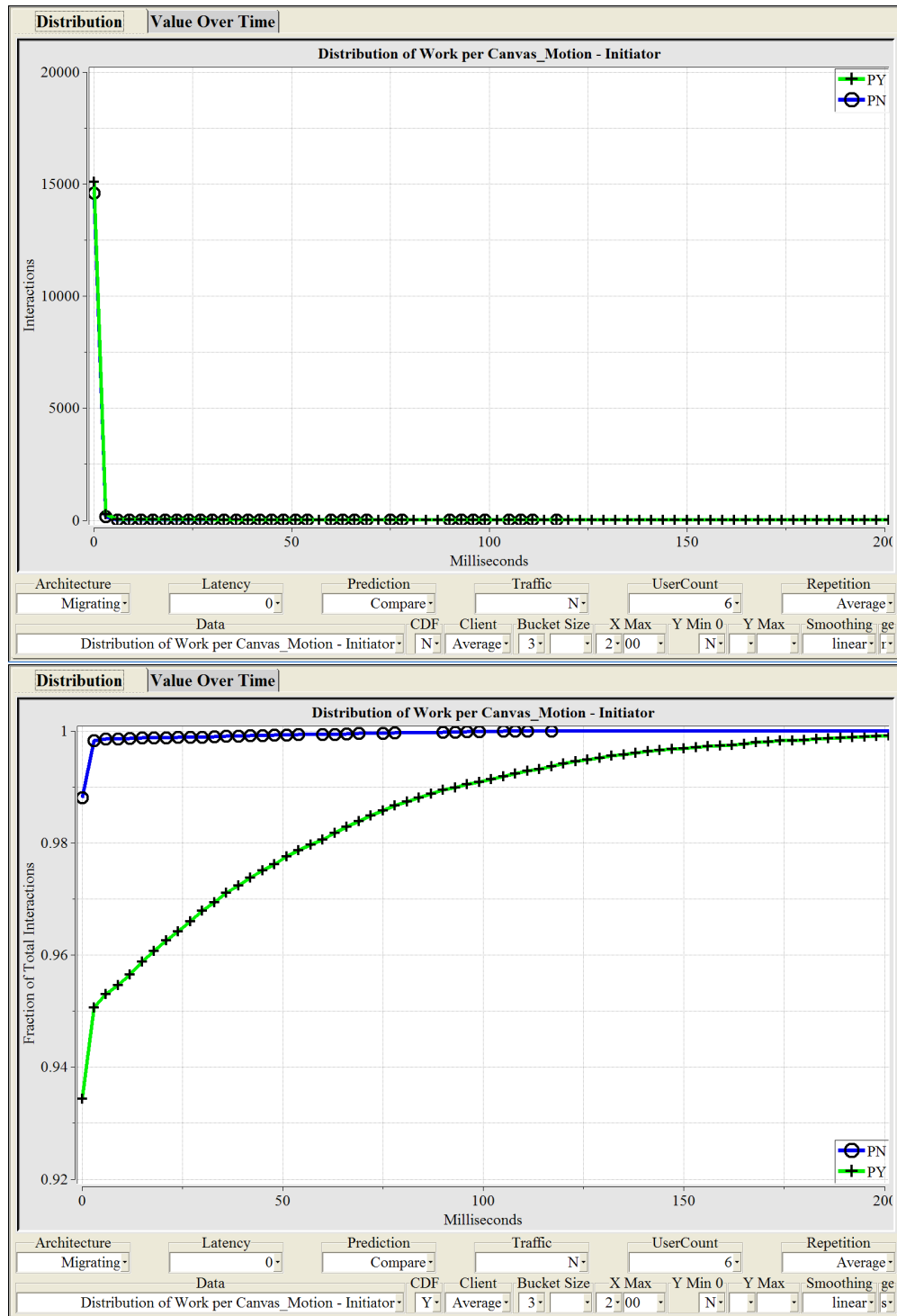


Figure 6.69: Distribution of Work Moving Cursor With and Without Prediction - Initiator

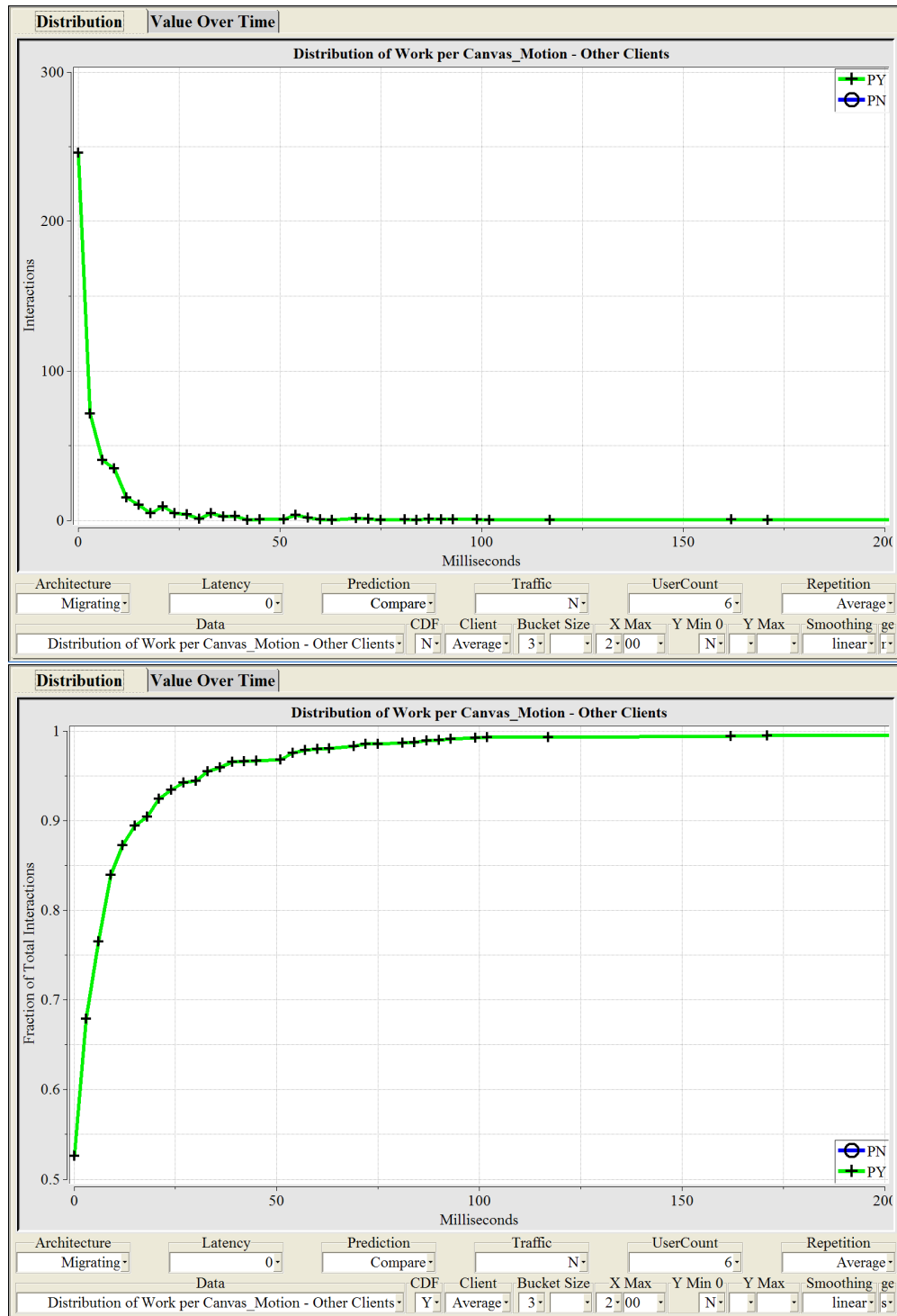


Figure 6.70: Distribution of Work Moving Cursor With and Without Prediction - Other Clients

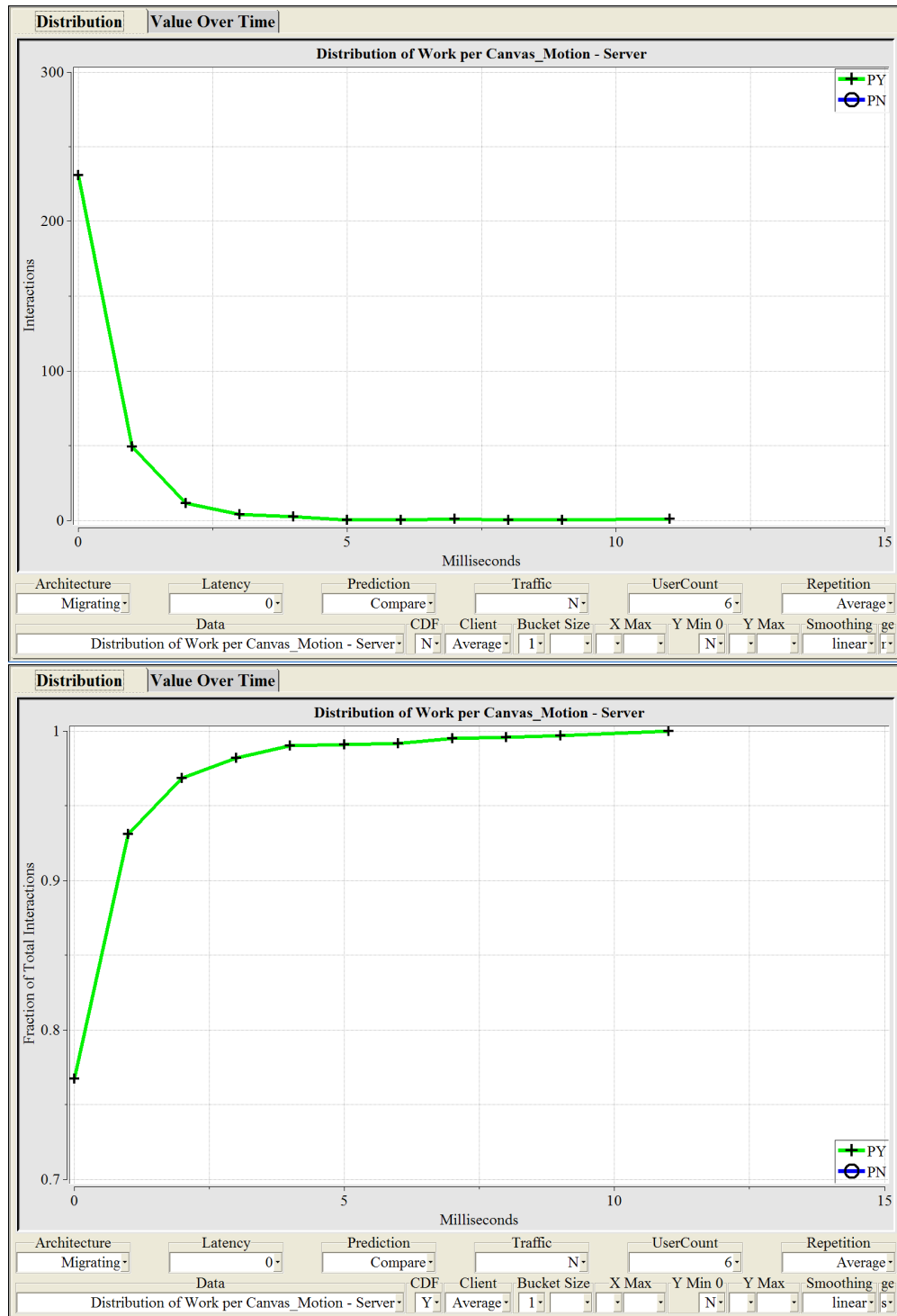


Figure 6.71: Distribution of Work Moving Cursor With and Without Prediction - Server

In sum, the costs in terms of unit of work of the migrating architecture are reasonable. While the experiments performed used a high percentage of the CPU on client machines, this should not be the case when the automated puzzle solver is not running and when client hosts of today and the future are used. In general, the server, which was run on a fairly contemporary computer, did very well in the experiments. Multi-threading should be explored for the migrating architecture, since current computers are multi-core.

Criterion 9: I will demonstrate the advantages of predictive migration based on telegraphed user intentions, made possible by Concur's fast migration times, rather than a past history of interaction.

This criterion raises a topic that was only tangentially explored in this dissertation, and will be further discussed in Chapter 7. The speed with which Concur migrates perspectives introduces the possibility of using better algorithms for determining when and to where migration should take place. Limited experiments in this dissertation showed promise for this technique, using only one migration prediction algorithm. The algorithm used was to predict a user's near-future interactions with a puzzle piece by analyzing the direction in which he is moving the cursor and determining which puzzle pieces were in that direction. The idea is that the piece he is likely to use can be migrated before the cursor even reaches it. Figure 1.28 demonstrated an advantage of this migration prediction algorithm over the commonly-used method of simply leaving an object where it was last used, in terms of the probability that the piece will already be migrated locally when an attempt is made to move it.

Chapter 7

Summary and Future Work

7.1 Summary

In this dissertation I have conducted an investigation of lightweight migration in support of centralized synchronous distributed collaboration. The primary aim was to achieve the superior performance characteristics of replicated architectures within the context of the simpler centralized architecture. The basic solution was a lightweight migration strategy. As a result of this investigation I have proposed a taxonomy of entities based on their migration characteristics. This taxonomy identified a number of broadly-useful entity types with good migration characteristics. I implemented an infrastructure supporting the migration of these entity types and the hosting of applications built around the entity taxonomy. I developed several applications representing a broad range of application types and demonstrated how such applications could be developed within a simple centralized framework. One of these applications was subjected to a large battery of experiments, which demonstrated performance similar to that of replicated systems.

An investigation like the one described in this dissertation invariably spawns new

ideas and raises more questions than it answers. Here are the ideas that I believe are most deserving of future investigative efforts:

- How might the ideas in this dissertation be applied to the Worldwide Web?
- How mature is the entity taxonomy developed in this dissertation? What new insights might be obtained by building a number of large applications around these entity types, and how might the taxonomy evolve as a result?
- Exactly how well do entities with lightweight migration characteristics map to desirable divergence opportunities? What is the best way to expose such opportunities to the user? What user mental model and implementation issues arise when you expose a wide range of divergence opportunities to users?
- Are continuously evaluated functions a new programming paradigm, or just a variant on old paradigms (e.g., constraints)? Is language support for such functions feasible and worthwhile?

These candidates for future work will be discussed briefly in the following section.

7.2 Future Work

In this section I will discuss a few ideas for future work that this dissertation has spawned.

7.2.1 Applying Concur to the Worldwide Web

When I initially began to undertake the work in this dissertation, I was motivated by certain difficulties that other engineers and I were encountering when attempting to provide richer, more interactive capabilities using Worldwide Web technologies on

the intranet at Hewlett-Packard. (I later encountered the same types of difficulties in my work at Amazon.com.) Having had a background in synchronous distributed collaboration, I realized that if these difficulties could be surmounted in web technologies, a side benefit would be that it would pave the way for synchronous distributed collaboration on the web, which has thus far been difficult to achieve in a way that is well integrated with the current web experience. (For a sampling of these efforts, see Artefact[BBD⁺98], CORK[IRC01], Groove[Gro05], LiveMeeting[Liv05], Promondia[GH97], Tango Interactive[BCF⁺], and WebEx[Web05].) The main reason for this is that a high degree of remote interactivity with the server is a requirement for synchronous distributed collaboration, since the results of one user's interactions must be promptly shared with other users.

Why is synchronous collaboration difficult to achieve with today's web? Primarily because of early fundamental design decisions with reasonable roots. The HyperText Transfer Protocol (HTTP)[BLFF96] was designed as a stateless request/response protocol, with requests initiated only by the client (browser), as shown in Figure 7.1. This design minimizes resource utilization in servers, since a server need only respond to isolated, individual requests from clients, without maintaining state about clients between requests. This, in turn, supports a very high degree of scalability, which is important in a context where thousands or millions of clients per day are just fetching documents.

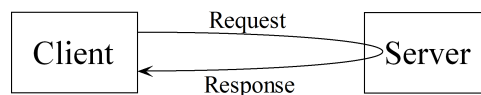


Figure 7.1: The HTTP Protocol

This is still a common scenario, but it is also common for today's web servers to present applications, rather than just document stores. Applications require applica-

tion state to be maintained between requests. If the amount of state is small, it can be maintained by the browser and passed to and from the server on each request and response. However, if the state is large, it is maintained on the server and indexed by a small amount of state maintained by the browser. Stateful sessions are thus commonly implemented on top of HTTP, negating the potential resource savings of the HTTP protocol.

Furthermore, some application classes require server-initiated messages to the browser (to avoid an unreasonable rate of polling by the browser). (See [PCH⁺00] for a more complete discussion of this topic.) This is particularly true of synchronous collaborative applications, which must immediately display the results of actions by one participant to all other participants. In recent years AJAX[AJA05] has been used to increase the remote interactivity of web pages. It accomplishes this with the standard HTTP protocol, by using JavaScript[Fla01]¹ to dynamically send multiple HTTP requests to the server in the context of a particular web page. This does indeed improve the interaction of a web page, but it does not really solve the problem of server-initiated messages to the browser.

Within the bounds of the HTTP protocol, the best that can be done is for the server to leave the connection open after sending the initial response, and to continue sending messages over this open connection (Figure 7.2). (Another way of looking at this is to view the server's response as taking a very long time to complete, and coming in segments separated by potentially long pauses.) Each message can be composed of JavaScript commands which are executed immediately on receipt by the browser, so that the user interface can change incrementally while the response is still outstanding. (This technique is also used to implement streaming audio and video

¹JavaScript is scripting code (not really related to the Java[AGH05] programming language) that can be delivered in an HTML page and run in a restricted environment in the browser.

over HTTP.) If the browser needs to send a message to the server while the response connection is still open, it can do so via nested² HTTP requests, invoked, e.g., by a Java applet³ embedded in the page (Figure 7.3).

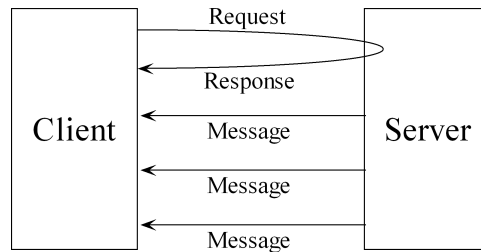


Figure 7.2: HTTP Streaming

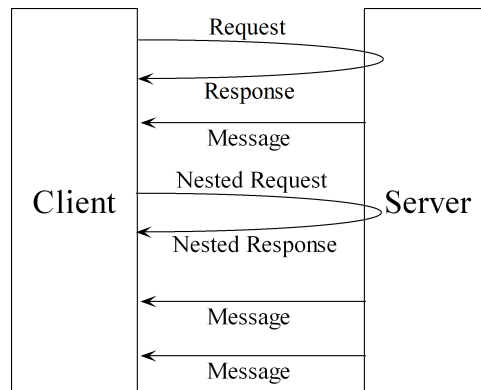


Figure 7.3: Nested Requests with Streaming

Staying within the bounds of HTTP is attractive, because it ensures that communications will not be impaired by a firewall⁴. But the technique described above

²There is no structural nesting of requests in HTTP. I just use this term to describe the fact that the nested request occurs while the outer request is outstanding, and that they both occur within the context of the same page (as with AJAX).

³A Java applet is a little Java program that, somewhat like JavaScript, can be delivered with an HTML page and run in a restricted environment in the browser.

⁴A firewall is software and/or hardware that filters network traffic in order to create a barrier protecting corporate or personal assets. Firewalls commonly block all or most connections into the protected domain, and may also block most connections initiated inside the domain to machines

is awkward, and inefficient for client-initiated messages. In addition, corporate networks appear to be moving toward technologies that enable outgoing connections to arbitrary ports (while enabling them to be monitored for security breaches), and personal firewalls often do not restrict such outgoing connections in the first place.

If we assume that outgoing connections through firewalls are not an issue, either because the web application is intended for intranet use, or for the reasons described above, there is a better solution using lower level, non-HTTP protocols (Figure 7.4). HTTP can still be used to fetch the initial web page, so that existing browsers can access the application. But thereafter, communication in both directions is carried out over a socket-level, full-duplex TCP/IP[Com00] connection back to the host from which the page originated⁵. Some applets use this approach, as would Concur.

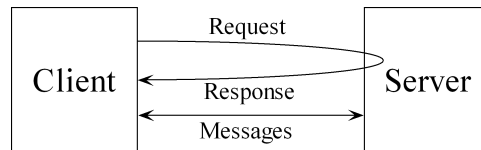


Figure 7.4: HTTP and Socket Connection

The HyperText Markup Language (HTML)[MK02], the historical markup language of web pages, presents its own set of problems for synchronous collaboration. The first problem is that HTML defines *what* should be displayed on a page, but does not specify exactly *how* it should be displayed, leaving browsers to make their own determination of how to display various constructs. Furthermore, the way in which

on the outside. Firewalls usually allow HTTP requests from inside the protected domain to the Internet as a whole. Thus, using HTTP as the only communication protocol typically avoids firewall restrictions.

⁵One of the restrictions in the applet environment is that an applet can only set up connections to the host from which web page originated. This is a security feature, designed to ensure that the user is not sharing information with a third party of which he is unaware. Concur would enforce the same restriction.

something is displayed depends on the size of the window, which may differ among users visiting the same (shared) page. Thus, the simplest and most straight-forward method of sharing, What-You-See-Is-What-I-See (WYSIWIS), is not supported by HTML.

As a means of increasing performance and scalability, HTML also encourages *autonomous interactions*, i.e., user interface changes that occur entirely locally, without any communication with the server. (Examples are the display of a drop-down list, the editing of text in a text box, button-press visualization, and highlighting of selected text.) These autonomous interactions cannot be (easily) shared, since they occur locally. JavaScript exacerbates the problem, since it can cause state and user interface changes locally in the browser.

Concur could support HTML sharing (but not WYSIWIS for HTML), just like any other UI technology. It could enable WYSIWIS sharing via embedded windows using user interface (UI) systems designed for desktop applications. In fact, the first version of Concur used Microsoft Internet Explorer as its client application, using a Browser Helper Object[Rob99] to gain access to the DOM document describing each web page, and embedding Tk windows within web pages.

Given the ubiquity and critical mass of acceptance attained by the web, any practical application of Concur would need to seriously consider delivering its capabilities in the web context. This could be part of a next-generation Worldwide Web experience. Collaborative capabilities are best presented as part of the underlying application fabric, rather than as separate “collaborative environments”, since this creates a common user experience across all applications, makes collaboration available to all applications, and facilitates transparent transitions among modes of work. One can imagine a future web experience where visiting the same URL necessarily implies sharing a web page WYSIWIS, and where various kinds of divergence from

WYSIWIS (also shareable) could be represented using the query strings⁶ of URLs.

7.2.2 Entity Taxonomy Maturity

The entity taxonomy presented in Figure 3.4 is new, and its use has only been explored with a few applications for this dissertation. It is likely that further experience with this taxonomy would uncover issues with the entities presented there and suggest new entity types for various purposes. The Timer Perspective was also not explored in the work of this dissertation. Further work is needed to apply various entity types to more and larger applications, and to develop the entity taxonomy as lessons are learned from this experience.

7.2.3 Divergence

This dissertation opens the door to a wider range of user-understandable divergence scenarios (from WYSIWIS to complete independence) than has been heretofore seen in collaborative systems. A taste of the possibilities was seen in Section 4.2.6. This dissertation really did not explore these possibilities in much depth. I have suggested and argued that the parameters to view functions map well to understandable divergence scenarios. Further work could solidify this relationship and explore such questions as how divergence possibilities might be presented to and communicated among users, and how they might be implemented and managed by infrastructures.

One can imagine future application development and presentation environments where developers present small items of functionality and combine them into sample applications which are then modified using common divergence mechanisms as users see fit. The applications so “developed” by end users could then feed ideas back into

⁶The query string of a URL is a list of name/value pairs appended to a URL.

more mature developer-produced sample applications that could be made available to a wider spectrum of users. In this way users would have more control over the applications they use, and might have a feasible process for influencing the evolution of applications for the good of the entire community.

One possibility is that applications would have a “divergence editor” mode, which would allow users to view divergence possibilities and effect divergence scenarios in a WYSIWYG⁷ fashion. That is, one might put an application into a graphical “divergence editor” mode, where a new controller would be assigned that would enable the user to browse for possible divergences and effect them. The application would remain “live”, but would be augmented by annotations describing possible divergences, and interactions with the application in this mode would effect changes in the set of shared and unshared components. For example, a user might be able to determine that he can detach scroll bars from a WYSIWIS page, such that he can scroll a document on his own. Any changes a user might make to an application in this way should be shareable in turn, e.g., via a new URL.

This work would necessarily require the investigator to perform user studies that would help him to understand the impact of divergence on the user’s mental model, and how one might present divergence possibilities in an understandable manner.

7.2.4 Continuously Evaluated Functions

Based on limited investigation by the author of this dissertation, continuously evaluated functions appear to be a novel and useful programming paradigm (albeit related to constraints). They could go a long way toward the development of robust, deterministic applications (both collaborative and non-collaborative) with good

⁷What you see is what you get.

mobility and latecomer characteristics. Further work should be initiated to explore language support for these functions, their performance characteristics, coding effort required, integration with other languages, and the applicability of this programming paradigm to various kinds of application development.

Appendix A

Experiment Automation with the Puzzle Solver

A.1 Puzzle Solver Motivation

My committee suggested that I automate my experiments rather than doing formal user studies. The advantages of this approach are:

- Many more experiments can be performed. I was able to perform a set of 384 experiments that took a total of 96 hours to run.
- Experiments can easily be re-run after code improvements. I re-ran my tests (or at least portions of them) many times before the final run.

The main disadvantages of this approach are:

- It can be difficult to prove that an automated test properly approximates how a collaborative application would be used by real human users. Fortunately, for the purposes of this dissertation, the main measure of interest was latency while dragging a puzzle piece, which is not highly susceptible to differences in how a piece might be dragged by different users.
- It does not allow for studies of how users accomplish tasks or how they react to different application features. Fortunately, this was not a goal of the present dissertation.

Automated experiments are often accomplished by tracing an actual set of experiments performed by human users, and then re-playing the users' actions for different points in the experiment space. My previous experiences with this type of replay software drove me away from this solution, for the following reasons:

- Timings are invariably different from run to run. This can make it very difficult to deterministically apply pre-recorded user actions to an application such that those actions make sense for subsequent replays.
- Application behavior may vary from run to run. This is particularly true when re-running a set of experiments against a changing code base, where code changes may cause different things to happen than were seen in the original live experiment.
- One would need to repeat the same random number sequences for each experiment run. This limits the code coverage, and opens one's results to the concern that they may have been unduly influenced by a particular set of random number sequences.

A.2 Puzzle Solver Overview

My solution was to create a smart controller that would emulate a user's thinking and actions. In this way, my puzzle experiments could be completely different every time, and could still be solved as if by a set of real human users. This is a time-consuming approach from the perspective of the developer, but once a such a smart controller is developed, it becomes a powerful tool that allows a large number of randomized experiments to be run, with a high degree of confidence in the results. A high-level description of the puzzle solver user interface and a list of its

main characteristics are given in 6.3. In this section I will give an overview of the implementation, and in the following section I will discuss its implementation in more detail.

Because I wanted to ensure that my latency measurements were accurate, I decided to initiate all actions as low-level user interface events (mouse movements, button clicks, etc.). This turned out to be a supported feature of the Tk toolkit. The puzzle solver was able to “see” his own and others’ interactions with puzzle pieces by monitoring the declarative user interface specification produced by the view function. Its decisions and subsequent interactions were determined by what was thus seen, just as a human user’s actions are determined by his observations of the puzzle pieces and their movements.

The collaborative solution of a puzzle is a long-term interaction composed of many lower-level interactions by multiple participants. Concur was implemented as a set of single-threaded processes, which made it impossible to directly maintain the context of a particular puzzle solver’s interactions in its own thread. As a result, my puzzle solver controller was implemented as a state machine, where the state of the machine encoded the task the solver was attempting to perform at the moment, along the whole spectrum from high to low levels of abstraction. For example, the solver might be attempting the following tasks at a given point in time, listed from high to low levels of abstraction:

- Working through a list of source pieces to attempt to match them to target pieces, starting with those nearest the solver’s favorite color, and working toward pieces increasingly distant from that color in a 3-dimensional color space.
- Attempting to match a particular source piece to a particular target piece.
- Moving the target fragment away from the edge of a table so that it can be

matched with the source fragment without putting any pieces off the table.

- Moving the cursor toward the target fragment.
- Moving the cursor a few pixels toward the target fragment.

Of course, there are many other states a solver might be in at a given moment.

When the machine is in a given state, it is looking for any of a set of events that might change its state. Here are some examples of such events:

- The puzzle was started. I choose a favorite color, and order the pieces to match based on their distance from that color in the 3-dimensional color space. I then begin attempting to match the first source pieces in this list.
- The piece I was moving moved as I expected it to move due to a drag operation I performed. If the piece has reached its destination, I will drop it. Otherwise, I will move it some more.
- The piece I was moving moved in a manner that I did not expect it to move in response to my drag operation. This means some other user was moving the piece at the same time. I will drop the piece, after which I will wait for a calculated backoff time before attempting to match a new source piece.
- A backoff timer expired. I can now continue by attempting the next match.
- I lowered a piece in the stacking order in order to look for a particular piece I wish to move. Another piece came to the top of the stack. If it is the piece I was looking for, I will pick up the piece in order to begin moving it. Otherwise, I will lower the new piece to the bottom of the stack.
- I just performed a snap operation and the source piece was successfully snapped to the target piece. I will move on to my next match attempt.

- I just performed a snap operation and the source piece was not successfully snapped to the target piece. This was the last target piece I want to attempt to match to this source piece. I will move the source piece to a random location and drop it, and then proceed with the next attempted match.
- The last piece of the puzzle was just successfully snapped. I will stop.

Each time an event is detected, the solver determines what state the machine it is in. Then, based on that state, it analyzes the event to determine whether or not it is interested in the event. If it is, the solver determines what state transition should be invoked on the machine. Finally, it invokes any actions associated with the transition, and sets the machine to the new state. Different levels of state abstraction are represented by a stack of states, such that when a low-level operation is complete, the stack pops up to the next higher level of abstraction. Operations at this higher level of abstraction are likely to push lower-level states on the stack. Error conditions can involve multiple pops of the stack, similar to how exceptions are handled in the runtime of a programming language.

Some time must usually elapse before a new operation is invoked (e.g., for the implementation of think times). Since we cannot pause the single-threaded process, these pauses must be implemented with timers, not sleeps. So the normal cycle of events is: set a timer, handle the timer event, invoke an operation, handle the resulting event(s), and repeat the cycle. Because one cannot always depend on one of a set of expected events occurring, a timer is almost always set before waiting for any other event. If the other event occurs before the timer expires, the timer is cancelled; otherwise a timeout exception transition occurs.

The overlapping mouse movements shown in Figure 6.58 further complicated the state machine implementation, because it required multiple outstanding drag opera-

tions to be in process at any given point in time. This set of overlapping drags was limited, in order to ensure that the user’s mouse did not get too far ahead of the piece. Once the window was full, the solver had to stop emitting drag operations. When a drag of a few pixels completed, it then needed to emit the next drag to fill the window. If an exceptional condition occurred, the solver would have to wait for the all events caused by outstanding operations in the window to clear before proceeding.

As you can see, the set of states and events needed for solving a puzzle is rather large. The puzzle solver is a complex piece of code that can only be effectively developed and debugged by running it against a large set of highly-randomized puzzle solutions. The fact that the solver could run 96 hours on up to 6 client hosts without a hiccup gives me confidence in its correctness.

The following section describes the puzzle solver algorithm in greater detail. Before getting into that, I want to note that, in retrospect, more attention could have been given to better match the automated puzzle solution to the solution of a puzzle by real human users. In particular, uniform distributions of delay times were used, where normal distributions would have better matched real users’ delays. The focus of the work in this dissertation was on studying low-level drag latencies (from the time a user moved his cursor a few pixels while dragging a piece, until the time he sees the piece actually move a few pixels in response). These measurements are not affected at all by the choice of delay distributions. However, more global measurements like task completion times could be affected by such choices.

A.3 Puzzle Solver Algorithm

In this section I will describe the puzzle solver algorithm in detail, while avoiding the actual implementation as a stack machine (which would be too complex). This

discussion will instead be a tour of the solver that covers all its important actions, decisions, and introduced delays.

When the server and all of the clients are ready, all solvers are simultaneously instructed to begin the solution of the puzzle. Each solver works through the same algorithm, as described below.

First a favorite color is chosen at random. This solver will begin by ordering all pieces¹ on the board by the distance in the 3-dimensional color space of their average color from this favorite color. The average color of each piece is pre-computed, and the distance is computed using the following formula:

$$\sqrt{(FavR - PieceR)^2 + (FavG - PieceG)^2 + (FavB - PieceB)^2} \quad (A.1)$$

The solver will successively choose the nearest remaining piece from this list, and attempt to match it to all other pieces previously so chosen. So initially it will choose the first piece and match it to the empty list of previously chosen pieces. It will then choose the second piece, and match it to the first piece. Next, it will choose the third piece and match it to the first two pieces (in an order described shortly), and so on. The use of a favorite piece and color distance ordering encourages the various “players” to work on different parts of the puzzle, i.e., pieces surrounding their favorite color.

When matching a new source piece to the set of previously tried target pieces, the target pieces are first ordered by how well the edges of the source and target pieces match in terms of the average colors of the halves of the pieces that are adjacent to the four edges that might potentially be matched. Figure A.1 illustrates this

¹A *piece* is one uniformly-sized (squarish) piece of the original puzzle. A *fragment* is a collection of such pieces snapped together.

computation. In this example, the source piece is being compared to the target

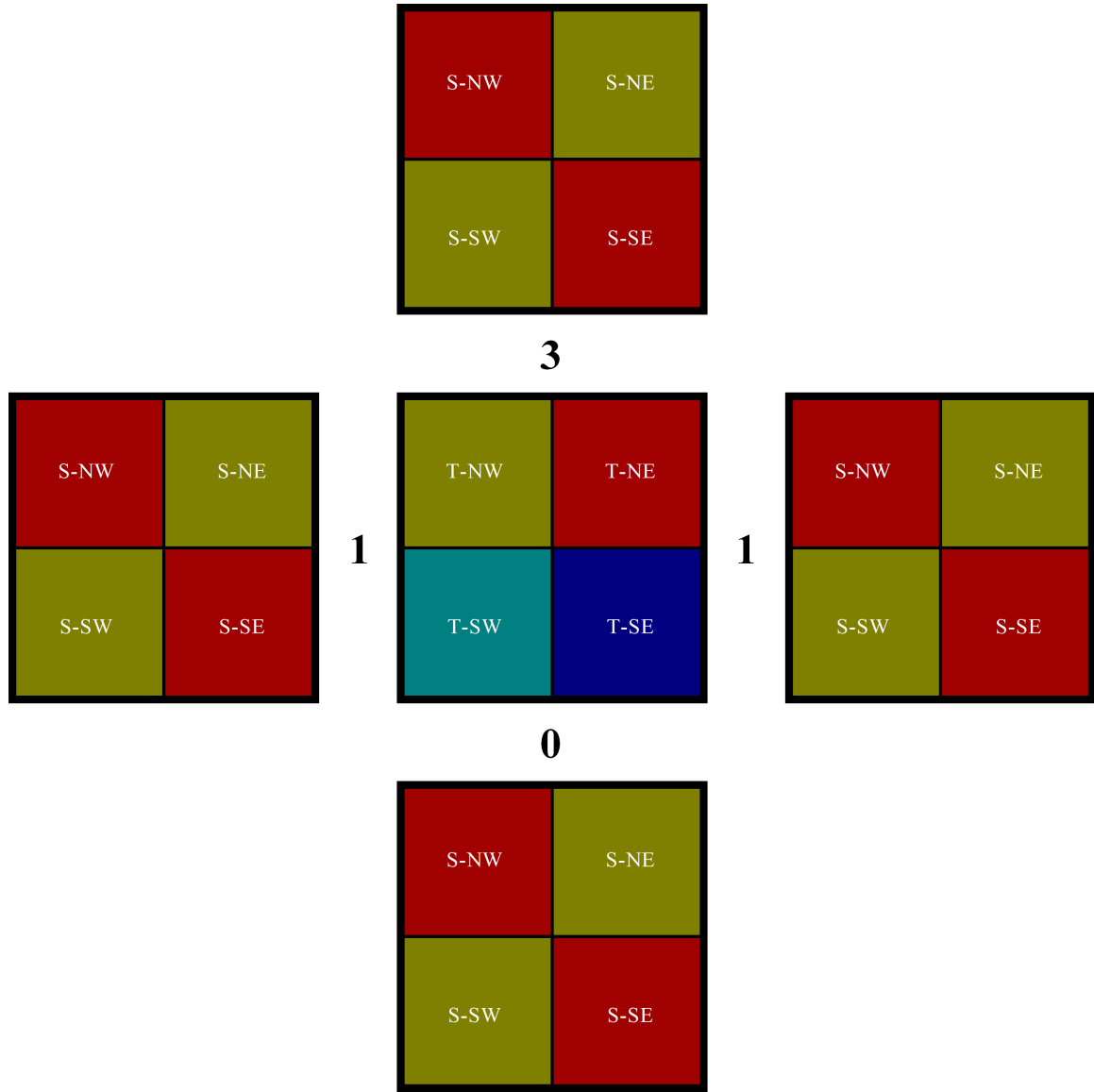


Figure A.1: Color Quadrant Matching Computation

piece along each of the four sides. Color averages are pre-computed for the four quadrants of each piece. These color averages are then made less specific by mapping each primary color component (Red, Green, and Blue) to 3 bits (8 colors). Thus, a highly-saturated blue would be 007, while a moderately-saturated yellow would be

440. Mapping to less specific colors allows us to use a set of 2^9 (512) color values and to compare for equality, rather than doing a more costly color distance comparison. (More sophisticated algorithms could be used, but this one was good enough for this work. It does tend to give a positive color score only to very good matches, however.) So, for example, quadrant S-NE is compared to quadrant T-NW, and quadrant S-SE to quadrant T-SW. If only one of the quadrants along an edge matches, that edge gets a score of 1. If two quadrants match, that edge gets a score of 3. The total score for matching the two pieces is the sum of the four edge scores. In the case of our example, the total score is 5. Giving priority to better color matches is what a human user would do, and it expedites the solution of the puzzle.

Once a source and target piece are chosen, the set of possible edges to match is computed. At most there will be four. Any straight edges, any already matched edges, and any pieces that do not match because of their shapes (two male, two female, an edge along one piece without a corresponding edge along the other, and a few more complicated cases) are discarded from consideration.

If any possible matching edges remain, an attempt is made to match the two pieces along each of those edges. It is here that actual visible operations on the puzzle begin to occur. It is also here that failures, delays, and aborted interactions begin to come into play. All delays in the system (for think times, etc.) are randomly chosen from a uniform distribution in the range of a specified number of seconds plus or minus $\frac{1}{2}$ second. This was good enough for the main target of this work, low-level drag latencies. If the puzzle itself had been the target of this work, distributions for think times and other delays would need to be chosen more carefully. The standard delay upon failure of any attempted operation is 3 seconds. Other delays are specified in the dialogue below.

In order to attempt a match of two edges, we must first pick up one of the pieces.

Normally the source piece is picked up and moved to the target piece. (Note that moving any piece implies moving the entire fragment to which that piece belongs.) However, if matching the source piece to the target piece would cause the source piece's fragment to be placed off of the board, we move the target piece to the source piece instead. If that would also cause the target piece's fragment to be moved off the board, we revert back to moving the source piece to the target piece, and we must move the target piece's fragment first so that there is room for the source piece's fragment on the board in its new location.

In any case, to move a piece we must first pick it up, and to pick up a piece we must first move the cursor to the center of that piece. We will next describe this cursor motion when a piece is not being dragged. Cursor positions are pre-computed from the current cursor position to the destination position in increments of 10 pixels. (That is, each incremental cursor movement would be 10 pixels if the cursor were being moved horizontally or vertically, or the same physical distance if being moved in any other direction.) This may seem a large incremental distance, but it was necessary to get reasonable cursor movement speed on slow computers². Then a cursor motion event is generated, causing the cursor to move a 10 pixel increment. When the solver receives the cursor motion event from the operating system, the event for moving the cursor the next increment is generated. No artificial delays were needed or introduced in this process; the cursor increment is the only parameter used to adjust cursor movement speed.

While the cursor is moving toward any piece, the solver watches for any movement of the piece toward which the cursor is being moved. If the piece is moved (by another

²In window systems, cursor motion is compacted during rapid movement by dropping intermediate cursor positions. This was avoided in the solver by introducing intermediate events into the event stream, to keep cursor movements from being compacted.

player), the operation is aborted. While the stack of operations at successively higher levels of abstraction could be popped to any of a number of levels, I chose to give up on the current source piece and move on to the next one whenever this happens. This is because if there is any conflict between players regarding the attempt to match pieces, it's probably best for one of the users to choose an entirely different source piece to match, to avoid future conflicts as much as possible. The delay introduced in this and all other cases of failure to accomplish something the solver tried to do is the standard 3 seconds. This is the simulated “think time” for a user to decide what to do next.

Once the cursor arrives at the center of the piece to be picked up, the solver must determine if the piece to be picked up is visible underneath the cursor, or if it is obscured by one or more pieces on top of it. If it is obscured, a button event is generated to lower the top piece. This is repeated until the piece to be picked up is at the top of the stack. Now the button event for picking up the piece is generated. It is at this point that piece migration is requested in the migrating architecture if prediction is not enabled. The solver will proceed to drag the piece before the migration occurs (if it ever does).

Dragging a piece is similar to moving the cursor. There are, however, a few more complications. The solver monitors the movements of the piece being dragged. If it is moved in a way that is inconsistent with the movements requested by this solver, the drag operation fails and the solver moves on to the next source piece. If the solver is dragging the piece toward another piece in order to attempt a snap operation, the solver similarly monitors the other piece for movements and aborts the operation if the other piece moves. Since dragging a piece takes more time than simply moving the cursor, we must also initiate a number of overlapping events for dragging the piece (Figure 6.58). It would be far too slow to move the cursor, wait for the piece to

move, move the cursor again, etc. But it would not do to have the cursor get too far ahead of the piece, either. The solver initiates a maximum of 10 incremental cursor movements. Once this window is full, it waits until a draw operation takes place, and then it fills the window again by generating another cursor motion event.

Once the piece is in place, it is dropped. If this is part of a larger attempt to match pieces, the event for a snap attempt is then generated. Once the snap attempt is complete, the solver moves on to the next match attempt.

Once an attempt to match the new source piece to all previous tried pieces is complete, the piece is moved to a random place on the board and dropped. If, however, the piece is part of a fragment of at least 5 pieces, this random placement of the piece is skipped. Thus, larger fragments tend to stay in one place, since their pieces have likely all been tried as sources, and if not, they are not randomly placed at the end of the match attempts of one of their pieces. Finally, a short pause (1 second) is executed, and the solver moves on to the next closest piece to its favorite color and begins matching it to previously-tried pieces.

The algorithm above does not guarantee that the puzzle will be solved once all solvers have finished one pass. When a solver finishes one pass, it then chooses a new favorite color and begins a new pass. This process continues until all of the pieces are snapped.

Appendix B

Exceptionally Long Latencies in the Puzzle Graph

One of my readers asked me why some of the latencies in some of the graphs (e.g., Figure 6.60) were so long (500+ ms). In response, I investigated the longest latencies in my experiments, and the results of that investigation are presented here.

Due to averaging of latencies across runs, the fact that graphs like Figure 6.60 do not cover all experiments, and truncation of graphs like Figure 6.60 on the right, it was not immediately apparent what the longest user-perceived drag latencies were. So I first queried my database to answer that question. The longest latencies turned out to be *very* long, around 13 seconds. There were 35 (Experiment, Client) pairs demonstrating latencies of 5 seconds or longer.

My database contained sufficient data to visually replay any experiment from any client's perspective. I wrote a tool to perform such replays for those portions of experiments where such long latencies were experienced.

Figure B.1 shows this replay tool in action. Along the top of the tool there is a time strip labeled with seconds from the beginning of the replay period. Red bars in this strip indicate where long latencies occurred, as found in the database. In this case, two such instances of long latencies are shown. (Usually there is not more than one per (Experiment, Client) pair.) A vertical bar in the strip shows the point in time currently being displayed in the puzzle below. In the figure, this bar is on the

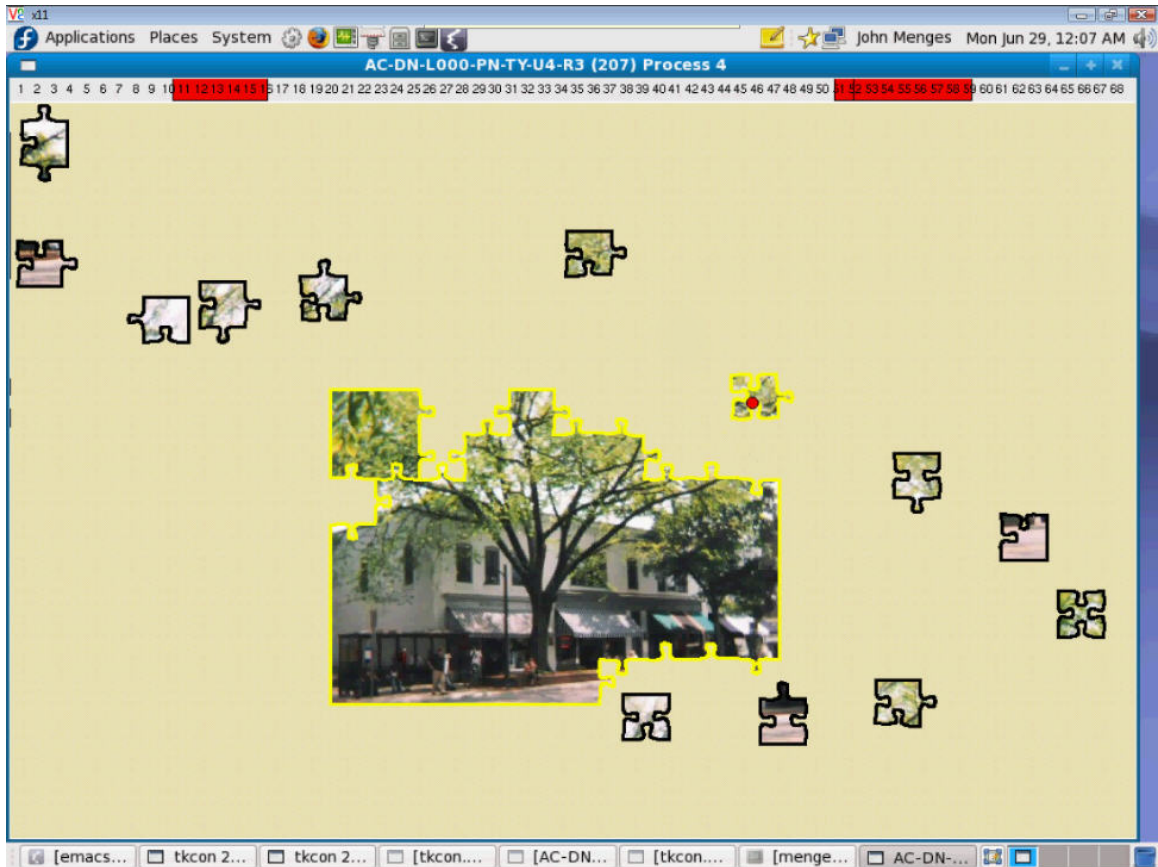


Figure B.1: Long Latency Replay Tool in Action

52 second mark, inside the rightmost red bar.

In the puzzle below, a dot indicates the position of the client's cursor. This dot is green when the user is not dragging a piece, and red when he is. As you can see in the figure, the dot is red, indicating that the user is dragging a piece. Therefore drag latencies are being computed for that user.

Next, note that a 4-piece fragment is being snapped by another user to the upper left corner of a much larger fragment. This turns out to be very significant. When two fragments are joined together, all pieces are consolidated into the fragment containing the upper-leftmost piece. At this stage of the puzzle solution, when a small piece is being joined to the upper-left corner of a large fragment, joining the fragments together takes substantial time. (Recall that the client machines on which this experiment were run were puny by today's standards.) Since the current user is trying to drag another piece at the same time, it gets abnormally-long latency measurements while the snap is taking place.

I viewed every one of the 35 latencies longer than 5 seconds in my 384 experiments, and in each case the scenario described above was the reason for the long measured latencies. Since snap time is linear with respect to the number of pieces in the lower right fragment, one could expect that many such long latencies below the 5 second threshold would be found, where the latency would taper off as that fragment size diminished. Thus, I am confident that this is at least the major source, if not the only source, of exceptionally long latencies affecting graphs such as Figure 6.60.

It is interesting to note that 5+ second latencies only occurred in the centralized and migrating architectures, not the replicated architecture. Further investigation of one of the above long-latency scenarios identified in greater detail the reason for the delay, and the reason it only affected the centralized and migrating architectures.

This is what happened. The user was dragging a piece that, for whatever reason

had not (yet) been migrated to his machine. This means he was effectively operating with a centralized architecture. In the centralized architecture, drag operations must go through the central server. (In the replicated architecture, the entire operation from event to draw is performed without communicating with the server.) The incremental drag operation in question was transmitted quickly to the server, which responded quickly with the corresponding perspective change. However, just prior to the server's handling this operation, the server serviced a request to snap two fragments together. This involves a large number of model changes within one transaction (moving pieces from the lower-right fragment to the upper-left fragment and changing their offsets to refer to the upper-left fragment). The replication of these changes caused a large number of messages to be sent from the server to the client, which then proceeded to effect the changes to its replica and execute the corresponding draw events. The feedback from the drag event was queued behind all of this other work.

One lesson from this analysis is that models should be designed such that wherever possible, single transactions do not involve a lot of changes to the model. By careful design of the model schema, snap operations could probably have been implemented as one or two changes to the model. Higher-level operations on trees could also be supported, such as the reparenting of all of the children of one node to another node in one operation, instead of one per child. These changes would vastly reduce the number of messages sent between Concur processes. However, unless the user interface supports multiple offsets for objects drawn on a canvas (Tk doesn't), the drawing operations might still take a long time, and some model operations may necessarily involve a large number of model changes.

So we are left to look at what might be done to allow interactions like dragging pieces to proceed with low latencies while unavoidable longer operations are taking

place. Given that the ordering of events between entities is not guaranteed even in Concur (Section 6.4), one solution might be to support more than one TCP/IP connection between processes. These connections are fairly heavyweight, so it would not be practical to have, say, one per entity. But it would be practical to have one for model operations and one for perspective operations. This would make use of Concur's natural distinction between these types. This would not only allow perspective operations (which are almost always smaller operations needing better latencies) to be transferred between processes without waiting behind a large number of model operations, but it would also allow the recipient of such messages to give higher priority to the servicing of perspective operations, or to interleave them with smaller units of model operations.

Finally, if single operations on models must take a long time to execute, the single-threaded model would not be adequate to support low-latency perspective operations in Concur. In this case, Concur should be enhanced to support multi-threading. This would be a useful enhancement in any case, given the predominance of multi-core computers today. Threads are lightweight enough that one could feasibly consider assigning a separate thread to each entity.

Bibliography

- [AGH05] Ken Arnold, James Gosling, and David Holmes. *The Java Programming Language*. Prentice Hall PTR, Upper Saddle River, NJ, 4th edition, August 2005.
- [AJA05] AJAX. Website, 2005. <<http://www.xml.com/pub/a/2005/02/09/xml-http-request.html>>.
- [AJJ⁺92] Paulo Amaral, Christian Jacquemot, Peter Jensen, Rodger Lea, and Adam Mirowski. Transparent Object Migration in COOL2. In Yolande Berbers and Peter Dickman, editors, *Position Papers of the ECOOP '92 Workshop W2*, pages 72–77, 1992.
- [AKSJ03] Jay Aikat, Jasleen Kaur, F. Donelson Smith, and Kevin Jeffay. Variability in TCP Round-trip Times. In *Proceedings of Internet Measurement Conference 2003*, Miami, FL, October 2003.
- [Ari08] A is A: Aristotle's Law of Identity, May 2008. <<http://www.importanceofphilosophy.com/Metaphysics.Identity.html>>.
- [AWF91] H. M. Abdel-Wahab and Mark A. Feit. XTV: A Framework for Sharing X Window Clients in Remote Synchronous Collaboration. In *Proceedings of Tricomm '91*, Chapel Hill, NC, April 1991.
- [AWJ94] Hussein M. Abdel-Wahab and Kevin Jeffay. Issues, Problems, and Solutions in Sharing X Clients on Multiple Displays. *Internetworking - Research and Practice*, 5(1), March 1994.
- [BBD⁺98] Jeff Brandenburg, Boyce Byerly, Tom Dobridge, Jinkun Lin, Dharmaraja Rajan, and Timothy Roscoe. Artefact: A Framework for Low-Overhead Web-Based Collaborative Systems. In *Proceedings of the ACM 1998 Conference on Computer Supported Cooperative Work (CSCW'98)*, pages 189–196, Seattle, WA, November 1998. Association for Computing Machinery (ACM).
- [BCF⁺] Lukasz Beca, Gang Cheng, Geoffrey C. Fox, Tomasz Jurga, Konrad Olszewski, Marek Podgorny, Piotr Sokolowski, Tomasz Stachowiak, and Krzysztof Walczak. TANGO Interactive - a Collaborative Environment for the World-Wide Web. White paper, Northeast Parallel Architectures Center, Syracuse University, Syracuse, NY. <http://trurl.npac.syr.edu/tango/Documents/Papers/WhitePaper/white_paper.html>.

- [BH93] T. Brinck and R.D. Hill. Building Shared Graphical Editors in the Abstraction-Link-View Architecture. In *Proceedings of ECSCW'93 (European Conference on Computer-Supported Cooperative Work*, September 1993.
- [BHR97] Joachim Baumann, Fritz Hohl, and Kurt Rothermel. Mole - Concepts of a Mobile Agent System. Technical Report TR-1997-15, 1997.
- [Bir98] Richard Bird. *Introduction to Functional Programming using Haskell*. Prentice Hall PTR, Upper Saddle River, NJ, 2nd edition, May 1998.
- [BLFF96] T. Berners-Lee, R. Fielding, and H. Frystyk. *RFC 1945: Hypertext Transfer Protocol - HTTP/1.0*. Internet Engineering Task Force, Network Working Group, May 1996. <<http://www.ietf.org/rfc/rfc1945.txt>>.
- [Bom08] Bomb Disposal, May 2008. <http://en.wikipedia.org/wiki/Bomb_disposal>.
- [BPSM⁺04] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, and François Yergeau. *Extensible Markup Language (XML) 1.0*. World Wide Web Consortium, 3rd edition, February 2004. <<http://www.w3.org/TR/2004/REC-xml-20040204>>.
- [CD96] Goopeel Chung and Prasun Dewan. A Mechanism for Supporting Client Migration in a Shared Window System. In *UIST '96: Proceedings of the 9th Annual ACM Symposium on User Interface Software and Technology*, pages 11–20, New York, NY, USA, 1996. ACM Press.
- [CD01] Goopeel Chung and Prasun Dewan. Flexible Support for Application-Sharing Architecture. In *Proceedings of the European Conference on Computer Supported Cooperative Work*, 2001.
- [CD04] Goopeel Chung and Prasun Dewan. Towards Dynamic Collaboration Architectures. In *CSCW '04: Proceedings of the 2004 ACM Conference on Computer Supported Cooperative Work*, pages 1–10, New York, NY, USA, 2004. ACM Press.
- [CDF08] Cumulative Distribution Function, July 2008. <http://en.wikipedia.org/wiki/Cumulative_distribution_function>.
- [CDR98] Goopeel Chung, Prasun Dewan, and Sadagopan Rajaram. Generic and Composable Latecomer Accomodation Service for Centralized Shared Systems. In *EHCI*, pages 129–147, 1998.

- [Chu02] Goopeel Chung. *Log-Based Collaborative Infrastructure*. PhD thesis, University of North Carolina Department of Computer Science, Chapel Hill, NC, 2002.
- [CJAW93] Goopeel Chung, Kevin Jeffay, and Hussein M. Abdel-Wahab. Accommodating Latecomers in Shared Window Systems. *IEEE Computer*, 26(1):72–74, 1993.
- [CJAW94] Goopeel Chung, Kevin Jeffay, and Hussein M. Abdel-Wahab. Dynamic Participation in a Computer-based Conferencing System. *Computer Communications*, 17(1):7–16, 1994.
- [CLO08] Common Lisp Object System, June 2008. <<http://en.wikipedia.org/wiki/CLOS>>.
- [Com00] Douglas E. Comer. *Internetworking with TCP/IP Vol.1: Principles, Protocols, and Architecture*. Pearson Education, Upper Saddle River, NJ, 4th edition, Jan 2000.
- [DC95] Prasun Dewan and Rajiv Choudhary. Coupling the User Interfaces of a Multiuser Program. *ACM Trans. Comput.-Hum. Interact.*, 2(1):1–39, 1995.
- [Dew99] Prasun Dewan. Architectures for Collaborative Applications. In Michael Beaudouin-Lafon, editor, *Computer Supported Co-Operative Work*, volume 7 of *Trends in Software*, chapter 9, pages 169–194. John Wiley & Son Ltd., New York, NY, February 1999.
- [Fla01] David Flanagan. *JavaScript: The Definitive Guide*. O'Reilly & Associates, Sebastopol, CA, 4th edition, December 2001.
- [GH97] Ulrich Gall and Franz J. Hauck. Promondia: A Java-Based Framework for Real-time Group Communication in the Web. In *Proceedings of the Sixth International World Wide Web Conference (WWW6)*, Santa Clara, CA, April 1997. World Wide Web Consortium. <<http://www.scope.gmd.de/info/www6/technical/paper100/paper100.html>>.
- [GHJV95a] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*, chapter 5, pages 293–303. Addison-Wesley, Reading, MA, January 1995.
- [GHJV95b] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, January 1995.

- [GM94] Saul Greenberg and David Marwood. Real Time Groupware as a Distributed System: Concurrency Control and its Effect on the Interface. In *Proceedings of the ACM 1994 Conference on Computer Supported Cooperative Work (CSCW'94)*, pages 207–217, Chapel Hill, NC, October 1994. Association for Computing Machinery (ACM).
- [GMU96] T.C. Nicholas Graham, Catherine A. Morton, and Tore Urnes. Clock-Works: Visual Programming of Component-Based Software Architectures. *Journal of Visual Languages and Computing*, pages 175–196, July 1996.
- [GR83] Adele Goldberg and David Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley, Reading, MA, 1983.
- [GR99] Saul Greenberg and Mark Roseman. Groupware Toolkits for Synchronous Work. In Michael Beaudouin-Lafon, editor, *Computer Supported Co-Operative Work*, volume 7 of *Trends in Software*, chapter 6, pages 135–168. John Wiley & Son Ltd., New York, NY, February 1999.
- [Gro05] Groove Virtual Office. Website, Groove Networks, 2005. <<http://groove.net>>.
- [GU92] T. C. Nicholas Graham and Tore Urnes. Relational Views as a Model for Automatic Distributed Implementation of Multi-User Applications. In *Proceedings of the ACM 1992 Conference on Computer Supported Cooperative Work (CSCW'92)*, pages 59–66, Toronto, Alberta, Canada, November 1992. Association for Computing Machinery (ACM).
- [GU96] T. C. Nicholas Graham and Tore Urnes. Linguistic Support for the Evolutionary Design of Software Architectures. In *ICSE '96: Proceedings of the 18th International Conference on Software Engineering*, pages 418–427, Washington, DC, USA, 1996. IEEE Computer Society.
- [GUN96a] T. C. Nicholas Graham, Tore Urnes, and Roy Nejabi. Efficient Distributed Implementation of Semi-Replicated Synchronous Groupware. In *Proceedings of the ACM Symposium on User Interface Software and Technology*, pages 1–10, Seattle, WA, USA, 6–8 November 1996.
- [GUN96b] T. C. Nicholas Graham, Tore Urnes, and Roy Nejabi. Efficient Distributed Implementation of Semi-Replicated Synchronous Groupware. In *Proceedings of the 9th Annual ACM Symposium on User Interface Software and Technology (UIST '96)*, pages 59–66, Seattle, WA, 1996. Association for Computing Machinery (ACM).

- [HBR⁺94] R. D. Hill, T. Brinck, S. L. Rohall, J. F. Patterson, and W. Wilner. The Rendezvous Architecture and Language for Constructing Multi-User Applications. *ACM Transactions on Computer-Human Interaction*, 1(3):81–125, June 1994.
- [HC06] Félix Hernández-Campos. *Generation and Validation of Empirically-Derived TCP Application Workloads*. PhD thesis, University of North Carolina Department of Computer Science, Chapel Hill, NC, 2006.
- [Hil92] Ralph D. Hill. The Abstraction-Link-View Paradigm: Using Constraints to Connect User Interfaces to Applications. In *CHI '92: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 335–342, New York, NY, USA, 1992. ACM Press.
- [How97] George A. Howlett. The BLT Toolkit. In Mark Harrison, editor, *Tcl/Tk Tools*, chapter 7, pages 265–342. O'Reilly & Associates, Sebastopol, CA, 1997.
- [IRC01] P. L. Isenhour, Mary Beth Rosson, and John M. Carroll. Supporting interactive collaboration on the Web with CORK. *Interacting with Computers*, 13(6):655–676, 2001.
- [Joh88] Robert Johansen, editor. *Groupware: Computer Support for Business Teams*. The Free Press (A Division of Macmillan, Inc.), New York, NY, 1988.
- [KP88] Glenn E. Krasner and Stephen T. Pope. A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80. *Journal of Object-Oriented Programming*, 1(3):26–49, August/September 1988.
- [Lib03] Jesse Liberty. *Programming C#*. O'Reilly & Associates, Sebastopol, CA, 3rd edition, June 2003.
- [Liv05] Live Meeting. Website, Microsoft, 2005. <<http://microsoft.com/livemeeting>>.
- [LJLR90] J. C. Lauwers, T. A. Joseph, K. A. Lantz, and A. L. Romanow. Replicated Architectures for Shared Window Systems: A Critique. In *Proceedings of the Conference on Office Information Systems*, pages 249–260, New York, NY, USA, 1990. ACM Press.
- [Lux95] Wolfgang Lux. Adaptable Object Migration: Concept and Implementation. *SIGOPS Oper. Syst. Rev.*, 29(2):54–69, 1995.
- [Mar02] Joe Marini. *Document Object Model*. Osborne/McGraw-Hill, Berkeley, CA, July 2002.

- [MK02] Chuck Musciano and Bill Kennedy. *HTML & XHTML: The Definitive Guide*. O'Reilly & Associates, Sebastopol, CA, 5th edition, August 2002.
- [MLC98] Dejan S. Milojcic, William LaForge, and Deepika Chauhan. Mobile Objects and Agents (MOA). In *Proceedings of USENIX COOTS'98*, Santa Fe, 1998.
- [Mor08] Electric Telegraph, May 2008.
<http://en.wikipedia.org/wiki/Electrical_telegraph>.
- [Nat08] National Climactic Data Center, June 2008.
<<http://www.ncdc.noaa.gov/oa/climate/research/ushcn/daily.html>>.
- [Nij00] Jan Nijtmans. Img Homepage, June 2000. <<http://members1.chello.nl/~j.nijtmans/img.html>>.
- [O'G98] Theodore Alan O'Grady. Flexible Data Sharing in a Groupware Toolkit. Master's thesis, Calgary, Alta., Canada, Canada, 1998.
- [Ous93] John K. Ousterhout. *An Introduction to Tcl and Tk*. Addison-Wesley, Reading, MA, 1993.
- [Pat90] John F. Patterson. The Good, the Bad, and the Ugly of Window Sharing in X. In *Proceedings of the Fourth Annual X Technical Conference*, Boston, MA, January 1990.
- [PCH⁺00] Joaquin Picon, Regis Coqueret, Andreas Hutfless, Gopal Indurkha, and Martin Weiss. *Design and Implement Servlets, JSPs, and EJBs for IBM WebSphere Application Server*, chapter 9. Vervante, August 2000.
- [PG99] W. G. Phillips and N. Graham. Software Architectures for Multiuser Interactive Systems. Technical Report TR-1999-425, Department of Computing and Information Science, Queens University, Kingston, Ontario, Canada, May 1999.
- [RG97] Mark Roseman and Saul Greenberg. Building Groupware with GroupKit. In Mark Harrison, editor, *Tcl/Tk Tools*, chapter 15, pages 535–564. O'Reilly & Associates, Sebastopol, CA, 1997.
- [Rob99] Scott Roberts. *Programming Internet Explorer 5*, chapter 12, pages 461–475. Microsoft, Redmond, WA, June 1999.
- [Rou03] Vassil Roussev. *Collaboration Transparency in Desktop Teleconferencing Environments*. PhD thesis, University of North Carolina Department of Computer Science, Chapel Hill, NC, 2003.

- [RV03] Matthew Robinson and Pavel Vorobiev. *Swing*. Manning Publications, Greenwich, CT, 2nd edition, February 2003.
- [SGR92] Robert Scheifler, James Gettys, and David Rosenthal. *X Window System: The Complete Reference to Xlib, X Protocol, ICCCM, XLFD*. Digital Press X and Motif Series. Digital Press, Bedford, MA, 3rd edition, February 1992.
- [Shn98] Ben Shneiderman. *Designing the User Interface: Strategies for Effective Human-Computer Interaction*, chapter 10. Addison-Wesley, Reading, MA, 3rd edition, 1998.
- [SJ95] B. Steensgaard and E. Jul. Object and Native Code Thread Mobility Among Heterogeneous Computers. In *SOSP '95: Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 68–77, New York, NY, USA, 1995. ACM Press.
- [Smi00] Chad Smith. *[incr Tcl/Tk] from the Ground Up*. Osborne/McGraw-Hill, Berkeley, CA, January 2000.
- [SSS99] Christian Schuckmann, Jan Schümmer, and Peter Seitz. Modeling Collaboration Using Shared Objects. In *GROUP '99: Proceedings of the International ACM SIGGROUP Conference on Supporting Group Work*, pages 189–198, New York, NY, USA, 1999. ACM Press.
- [Str00] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, MA, special 3rd edition, February 2000.
- [SWND03] Dave Shreiner, Mason Woo, Jackie Neider, and Tom Davis. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 1.4*. Addison-Wesley, Reading, MA, 4th edition, November 2003.
- [UN94] T. Urnes and R. Nejabi. Tools for Implementing Groupware: Survey and Evaluation, 1994.
- [Urn92] Tore Urnes. A Relational Model for Programming Concurrent and Distributed User Interfaces. Master's thesis, Norwegian Institute of Technology, University of Trondheim, April 1992.
- [Urn98] Tore Urnes. *Efficiently Implementing Synchronous Groupware*. PhD thesis, Department of Computer Science, York University, Toronto, Ontario, Canada, 1998.
- [Web05] WebEx. Website, Microsoft, 2005. <<http://webex.com>>.
- [Web06] Webopedia Definition of Semantics in Computer Science, May 2006. <<http://webopedia.com/TERM/s/semantics.html>>.

- [WG02] Eric White and Chris Garrett. *GDI+ Programming: Creating Custom Controls Using C#*. Peer Information, Hoboken, NJ, June 2002.
- [WH03] Brent B. Welch and Jeffrey Hobbs. *Practical Programming in Tcl & Tk*. Prentice Hall PTR, Upper Saddle River, NJ, 4th edition, June 2003.
- [Xer08] Xerces2 Java Parser. Website, The Apache XML Project, 2008. <<http://xml.apache.org/xerces2-j>>.