Managing Collaboration in the nanoManipulator

Thomas C. Hudson, University of North Carolina at Wilmington, hudsont@uncw.edu
Aron T. Helser, 3rdTech, Inc., helser@3rdtech.com
Diane H. Sonnenwald, University of Borås, Sweden, Diane.Sonnenwald@hb.se
Mary C. Whitton, University of North Carolina at Chapel Hill, whitton@cs.unc.edu

Abstract

We designed, developed, deployed, and evaluated the Collaborative nanoManipulator (CnM), a distributed, collaborative virtual environment system supporting remote scientific collaboration between users of the nanoManipulator interface to atomic force microscopes. This paper briefly describes the entire collaboration system, but focuses on the shared nanoManipulator (nM) application. To be readily accepted by users, the shared nM application had to have the same high level of interactivity as the single user system and include all the functions of the single user system. In addition the application had to support a user's ability to interleave working privately and working collaboratively. Based on our experience developing the CnM, we present: a method of analyzing applications to characterize the concurrency requirements for sharing data between collaborating sites, examples of data structures that support distributed collaboration and interleaved private and collaborative work, and guidelines for selecting appropriate synchronization and concurrency control schemes.

Keywords: state replication, collaborative virtual environments, scientific collaboration, distributed collaboration, concurrency control, nanoManipulator

1. Introduction

In a multi-year, NIH-funded project designed to discover whether collaborators working remotely can perform "good" science, we designed, developed, deployed, and evaluated a collaboration system that supports distributed, shared use of the nanoManipulator (nM) over public network connections. The nM is an interactive interface to atomic force microscopes (AFM) developed at the University of North Carolina at Chapel Hill. (Taylor and Superfine 1999; Sonnenwald, Bergquist et al. 2001) The nM provides the user with 3D visualizations of data from the microscope and a force-feedback device that allows users to feel and modify the sample in the microscope. The nM is a desktop virtual environment (VE) system in the sense that (nanoscale) features of the sample are scaled up roughly one million times and made visible and tangible to the user via the nM interface. Our collaboration system can be considered either a collaborative virtual environment (CVE) where the users are not co-located or a distributed virtual environment (DVE) where the users are collaborating. We will variously refer to it as being a CVE or a DVE when discussing it with respect to other systems, since the literature has used these terms separately but somewhat interchangeably.

To build the Collaborative nanoManipulator (CnM), we modified and added features to the nM application itself and integrated it with system components supporting communication and shared application functionality between remote users. The Collaborative nanoManipulator system is now in commercial and academic use and has been the subject of both a controlled experimental evaluation and an ongoing ethnographic evaluation. This paper briefly describes the entire collaboration system, including the hardware and software, but focuses on the design of the shared nM application.

Three requirements drove the architecture of the shared nanoManipulator application. One, it had to be highly interactive, with both response time and user-to-user time less than 300 ms. Two, it had to support the ability for a collaborator to interleave periods of working independently (private mode) with periods of working collaboratively (shared mode), with the associated requirement that there be an easy mechanism for copying application state data back and forth between the independent (private) and collaborative (shared) work modes. Three, it had to have all the features and functionality of the single-user system; in practice; this was a corollary of the interactivity requirement. (Work is ongoing to improve a remote user's force-feedback experience, but that work is beyond the scope of this paper.)

The interactivity and independence requirements both contributed to the early design decision to implement the collaborative system with replicated copies of the application in each user's computer. Were the system implemented with a single, central copy of the application, any user not collocated with this single server would have all their input actions slowed by network delay. Replication is a necessary precondition of allowing all users to see the effect of their input without adding any network latency to the system response time. Local replicas of the application and careful design of concurrency control also give us the desired property that when in private mode the user experiences the same performance as when running the single-user application.

Once this fundamental architectural decision was made, we addressed the problems of (1) characterizing the control parameters in our system so that we could match them to appropriate synchronization and concurrency control mechanisms, and (2) creating and managing the two sets of state data required at each site to support both shared and private work. We found that an extension of the model-view-controller paradigm (Krasner and Pope 1988) improved our understanding of the system, leading to a better architecture and improved maintainability.

In this paper we present a set of guidelines for selecting an appropriate concurrency control technique for the various types of VE system data, present an extension of the model-view-controller paradigm to help developers of collaborative VE systems analyze applications and characterize control parameters, and describe data structures that support each user's ability to move easily between shared and private work modes.

2. Background

This section provides background in performance requirements to support interactivity, reviews concurrency control techniques, takes a further look at concurrency control issues in distributed virtual environments, and concludes by comparing the collaborative nanoManipulator to other scientific collaboratories.

2.1 Interactivity Requirement

In addition to application-specific functionality, chief among any set of system requirements is usability. Latency is a significant factor determining the usability of an interface. Graphical and VE applications have particularly stringent latency requirements because their interfaces are *interactive*: users directly manipulate parameters controlling the images they see using continuous input devices such as mice or, in the case of the nM, a Phantom ™ force feedback device. The usability of interactive interfaces degrades significantly when visual feedback is not essentially immediate. Cycles of overshooting and undershooting the desired value can result when the (impatient) user makes another input before seeing the system's response to the previous input.

Collaborative applications must address two distinct types of latency: response time and user-to-user time. (Bhola, Banavar et al. 1998) *Response time* is the time between a user's input and that user seeing the results of his input. *User-to-user time* is the time between a user's input and a remote collaborator seeing the results of that input. Different aspects of the system design contribute to these two latencies; both must be minimized to create a high-performance VE system.

Response times must be tightly bounded in interactive interfaces. The VE community has reported 100 ms as an approximate upper bound for direct manipulation tasks. (Macedonia, Zyda et al. 1994; Ware and Ralakrishnan 1994; Bryson and Johan 1996) In the context of a two-dimensional collaborative application, Bhola *et al.* state that the limit varies between 50 and 100 ms (Bhola, Banavar et al. 1998). When, as in force-feedback systems, user input is driving a control loop ("human-in-the loop" control systems), response time becomes even more critical: 50 ms of latency in flight simulators reduces performance. Latency can cause mechanical system instability (Wickens and Baker 1995) if response is so slow that operators induce oscillations by sequentially and repeatedly over- and under-shooting the desired control setting. Although operators in some cases can be trained to be effective in the face of latencies on the order of one second (Taylor, Chen et al.) we do not want to impose such an impediment on our users.

When collaborating users are attempting to coordinate their actions, or understand how their actions affect one another, the user-to-user time is important. Several authors report that 200 ms of user-to-user delay interferes with the completion of closely coordinated tasks in collaborative virtual environments (Vaghi, Greenhalgh et al. 1999).

Our experience shows latency to be less a constraint in our DVE than reported by these other researchers. Our users have been tolerant of response times near 200 ms. System latency in excess of 200 ms interferes with haptics, but not (according to anecdotal reports) with use of the other facets of the system. Only when latency approaches 400-500 ms have our users considered the system unworkable.

Based on this prior work, we established the following targets for our latencies. For response time, we hope to match or improve on the current nM's single-user response time of approximately 200 ms. For user-to-user latency, we established a target 250 ms, recognizing that we might not achieve that goal due to use of shared public networks. The users' comments on the system and their user of it are our final determinants of acceptable latencies.

2.2 Concurrency Control

Users of a distributed shared application expect to see an application state identical to that seen by their collaborators. Typically, application designers assume that all collaborators need to have completely consistent views to get any work done. Most users, becoming aware that they are seeing different application states, cease to trust that the system is faithfully communicating the "world" they are sharing with their collaborator and the collaboration breaks down. However, in CVEs, as in more typical office productivity or software development applications, users may actually be quite productive with a relaxed degree of consistency so long as they aren't aware of it.

Concurrency control algorithms embedded in the application are used to guarantee that all users see a consistent application state. A fundamental idea in concurrency control is to *serialize* inputs to the application coming from the multiple sites so that the inputs are executed in the same order at each site. If the input sequences are the same at each site, each user will see the same application state and maintain confidence in the system.

The remainder of this section discusses requirements for concurrent control mechanisms, then reviews several common techniques. The assumption throughout is that there are multiple simultaneous users of the shared application. While the concurrency control discussion also applies to applications which are shared asynchronously, our system is designed to support concurrent users, so meeting the needs of that type of use is our major concern.

Concurrency Control Requirements There are three requirements for concurrency control: do not allow application behavior that causes irreversible damage to data or equipment, do not cause the user see an unexpected application state, and do not degrade the user's experience. We took it as our goal in the CnM to meet the first of these requirements and to make application-appropriate tradeoffs in addressing the second and third. Our overall goal is to maximize adherence to these requirements while incurring as little disruption to the users' workflow and cognitive activity as possible.

The first requirement for concurrency control mechanisms is an absolute: the system must prevent application behavior that may cause harm to data or devices (Herlihy 1987) In the CnM system, the expensive atomic force microscope can be physically harmed by improper sequencing of control commands, so we had to implement concurrency control mechanisms that prevent such an occurrence.

The requirement that users always see an expected application state is sometimes expressed as the *Principle of No Surprises*, or the *Isolation Property* (Weihl 1993). Ideally, users should never be surprised by the application's response to their input or see unintended effects due to a collaborator's simultaneous actions.

The latency requirements of interactive graphical applications make the time consumed by concurrency control a major issue in effectively sharing data between geographically separated users. As is described more fully in later sections, we made tradeoffs in system design to maximize the users' interaction experience while also meeting concurrency control requirements. Our general approach is to keep latency low and interactivity high at the cost of allowing users to (occasionally) violate the principle of no surprises. If two users simultaneously attempt to change the same parameter, one user might see their change occur, only to be replaced by the other user's change a few frames later (typically less than half a second).

Achieving Concurrency—Locks. One way to achieve serialization is to employ locking mechanisms so that only one user at a time has access to a particular item of data or control input. If a shared application has centralized access control and employs locks for concurrency control, all clients suffer delay of one network round-trip time on every operation. To speed up client operations, state can be replicated at every client's computer. Clients can then quickly read local state, but need to execute a distributed concurrency control protocol to obtain and release locks when writing to their own copy of the application's state and to disseminate the results of writes to other users. Locking mechanisms differ in two fundamental ways; they can be *coarse-grained* or *fine-grained* and they can be *implicit* or *explicit*.

Floor control is a coarse-grained, explicit locking technique and is the simplest implementation of database-style locking in collaborative applications. Floor control uses a single lock to guard access to the entire application or document being shared (Malpani and Rowe 1997). Only one user at a time can actively manipulate the application. Inactive users explicitly signal, e.g. press a button, to request control of the application. As in a well-run meeting, only one speaker at a time "has the floor" and can address others or take action, while others watch and listen. Microsoft's NetMeeting™ is one familiar product that uses floor control.

If an application is divided into parts and access to each part is controlled by a separate lock, the locks are said to be fine-grained. For example, a word processor document might be divided up into many pieces with one lock controlling each piece; only one user can manipulate a given section of a document, but users may work in parallel on different sections of the document. Fine-grained locks increase the need to interleave lock management with the user's natural workflow, possibly breaking concentration on the task and usually giving the user a worse experience than the single-user application.

Implicit requests for a lock are an attempt to decrease workflow interruption. Rather than explicitly requesting a lock, the user simply begins a manipulation; the system then checks to see if that part of the document is locked, and if so aborts their operation. While implicit locks avoid the workflow disruption of explicit locking, they increase application latency, and can cause an unexpected loss of work when operations are aborted, violating the “no surprises” directive and discouraging users.

A system using implicit locks can add additional complexity to attempt to reduce serialization latency. If locks “migrate” to user systems – that is, for each lock, one user’s computer is the arbiter of the lock state (instead of a particular lock server) – migration can be controlled to minimize expected latency. When a user obtains a lock on an object, that object’s lock (or ownership) migrates to the user’s computer. Subsequent attempts to lock that object by that user avoid network latency penalties, until another user uses it. Thus, like the concurrency control methods we will discuss subsequently, migrating locks work best in the absence of contention.

To avoid the round-trip networking latency incurred with input serialization via locking, we looked to other concurrency control techniques.

Achieving Concurrency—Optimism. Optimistic concurrency control is a class of algorithms that achieves application consistency with minimal latency. Without locking, latency is primarily a function of the time required to serialize the flow of commands for each application site, rather than being a function of round-trip network delay. Optimistic concurrency control has to be careful to preserve the intention of the multiple concurrent users of the system.

Following Amdahl’s Law — *make the common case fast* — systems using optimistic concurrency control perform an operation locally, then broadcast the change to other users. At the remote sites, the operations are compared to see whether they conflict with simultaneous operations on other concurrently-running instances of the application. If conflicts occur, an application-dependent procedure resolves them at each site. (Herlihy 1990) Since most operations do not cause conflicts, they can be executed rapidly, without the latency penalty of locking mechanisms.

Many groupware systems use optimistic concurrency control because of their sensitivity to latency. Recent work has extended optimistic approaches from text editors (Ressel, Nitsche-Ruhland et al. 1996) to spreadsheets and 2D graphics editors. The work reported in this paper further extends them to an interactive 3D graphics application. Control of view and visualization parameters in CVEs is an especially appropriate application of optimistic concurrency controls because the simple consistency requirements of these parameters mean that optimism can be implemented without sophisticated contention resolution protocols.

2.3 Concurrency Control in Distributed Virtual Environments

Visualization of scientific data is one of a number of VE applications in which one or more users explore a virtual world model. While in scientific visualization VEs, users may vary their viewpoints, vary the viewing parameters, use analysis tools, and generate annotations and other supplemental data. The nM supports all these activities and the CnM must also support them in a way that maintains application consistency between sites. In some VEs, the world model is static, i.e. neither the users nor the application modify most of the data during the VE session. However, in other VEs, the world model is dynamic: the application may allow the user to interact with and modify the data, or the world model may be generated from a real-time data stream. Such data may come from a real-time sensor, be generated by a simulation, or be the playback of a recorded sequence from one of those sources. The nM supports dynamic data of both these types and the CnM must also. The primary source of world model/data in the nM and CnM is a live stream of data from the AFM (or a playback of such a stream of data) and, during a live session, the user can modify that world model data by modifying the sample in the microscope.

Meehan (Meehan 1999) surveyed multi-user distributed virtual-environment (DVE) systems, showing the wide range of concurrency control approaches that have been tried in distributed, shared virtual environment applications. Concurrency control performance is dependent on the communication speed between distributed sites and the communication speed, in turn, depends on both network bandwidth, the network's capacity to carry data, and network latency, the inherent network propagation and queuing delays. Depending on the amount of data that needs to be transferred between sites, applications may find one or the other of these network characteristics to be their bottleneck.

Massively multi-user distributed systems like NPSNET and AVIARY are examples of DVEs with a serious performance constraint in network bandwidth. These systems were designed to maintain application consistency while minimizing the amount of data transferred between distributed users and also avoiding the latency penalty of locking protocols (Snowdon and West 1994; Macedonia, Brutzman et al. 1995).

In the shared nanoManipulator application, with its relatively small amounts of data to be transferred, the network's latency is a much larger component of user-to-user latency than is network bandwidth. Propagation and queuing times are much greater than transmission time; distance and network congestion contribute more to user-to-user latency than does bandwidth. In this work, we assumed that network latency is beyond the application's control. Instead of trying to control network performance, we change the implementation of the application to reduce its demands of the network – i.e. we try to improve application-level performance for a given network performance. Although not used in our system, another technique for reducing the impact of the network latency is to modify the application's user interface in a way that helps users adapt to latency. A good survey of this approach can be found in. (Vaghi, Greenhalgh et al. 1999).

2.4 Scientific Collaboratories

Our project is one of eight funded by the NIH to evaluate collaboration technology. Scientific collaboratories most frequently fall into one of two classes: systems that allow single-user access to expensive, rare instrumentation such as the Telescience Portal project within the National Center for Microscopy and Imaging Research at the University of California San Diego (NCMIR, 2003) and systems that allow shared access to large data repositories, such as the Protein Data Bank maintained by the Research Collaboratory for Structural Bioinformatics (Berman, Westbrook et al. 2000).   The CnM is more like the former than the latter, but with differences we discuss later.

Another example of a scientific collaboratory providing shared access to remote equipment is the Upper Atmospheric Research Collaboratory (UARC). UARC provides simultaneous access for scientists all over the world to a number of widely dispersed scientific instruments that observe Earth's ionosphere (Olson, Atkins et al. 1998). Another is Bugscope (Potter, Carragher et al. 2000), a tool for K-12 teachers. Bugscope, one of several notable systems implemented at the Beckman Institute, is an application that lets remote users share a Scanning Electron Microscope (SEM).

The Beckman Bugscope system is typical of remote instrument access collaboratories in that is has explicit floor control, i.e. one user has control at a time (though others may be passive observers), and it is not highly interactive. Users have a web-based interface that shows them one frame of video captured by the SEM. Buttons on this interface cause the SEM to pan, zoom, or otherwise modify the SEM control parameters in discrete steps. Similarly, the user interfaces to UARC are a command line or a collection of 2D widgets, manipulated without immediate feedback from the application. In web-based interfaces like these in which latency cannot be controlled, humans tolerate a rather amazingly high level of latency.

Neither of the above scientific collaboratories is an interactive application in the sense we use to describe the CnM, i.e. they do not accept continuous streams of input data and provide almost instantaneous visual feedback when parameters are changed. When used by a remote scientist and a local AFM technician, the CnM is a scientific collaboratory system that allows single-user remote access to an instrument. However, the goal in the design of the system was to provide support for concurrent use by collaborating scientists—two or more individuals engaged in a shared intellectual endeavor. More specifically, the design goal was to support scientists working together to perform a live experiment on the AFM using the shared nanoManipulator application or working together to perform analysis of data collected previously using both the shared nM application and shared analysis and productivity tools. In a remote instrument access collaboratory, the communication job is over when the data has been properly gathered on the instrument and accurately transmitted to the remote scientist. In the CnM, that is just the beginning, as it must also support interactive replay of data collection and interactive analysis procedures for two or more distributed collaborators.

3. An Overview of the nM and the Collaboration System

The nanoManipulator. The single-user nanoManipulator is a software and hardware tool that provides interactive 3D visualization of data from an Atomic Force Microscope and gives a user force-feedback control of the microscope's tip. The application allows scientists to see, feel, and modify experimental samples ranging from DNA and carbon nanotubes to viruses and cells. All streams of data coming from the microscope are recorded and the system can replay the data; this allows scientists to (re)analyze experiments (Taylor and Superfine 1999).

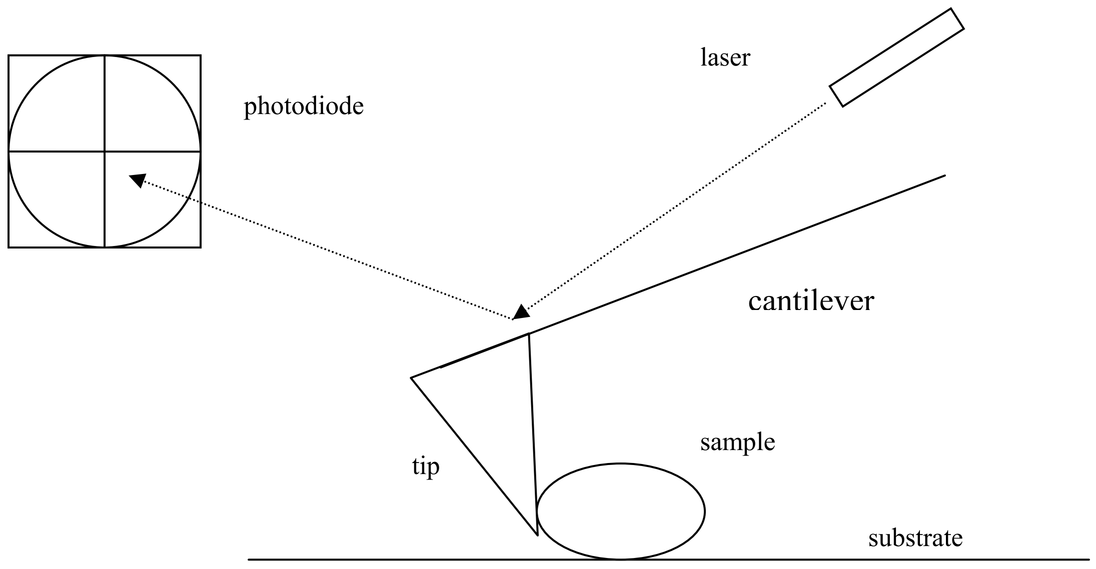


Figure 1 Schematic design of an Atomic Force Microscope.

An Atomic Force Microscope is a sophisticated instrument that allows scientists both to determine the shape and characteristics of and to exert force on microscopic samples as small as an individual virus (c. 1000 nm) or strand of DNA (c. 3 nm). This sample is placed on a substrate; a long, sharp tip (typically narrowing down to a 10 nm radius) attached to a flexible cantilever is scanned in a raster pattern back and forth over the substrate. A laser bounces off the back of the cantilever onto a photodiode to determine how much the cantilever is bent; given the cantilever's spring constant, this determines the amount of force being exerted on the tip. If the force is kept constant, the tip traces out the topography of the surface, similar to a relief map. The AFM can also increase the force on the tip to modify the sample, with applications such as cutting a strand of DNA, squeezing a virus, or moving a gold colloid into a narrow gap in a wire (Taylor and Superfine 1999; Leckenby 2000). The primary output of an AFM is a height field, which is rendered as a 3D surface in the main graphics window of the nM (Figure 2).

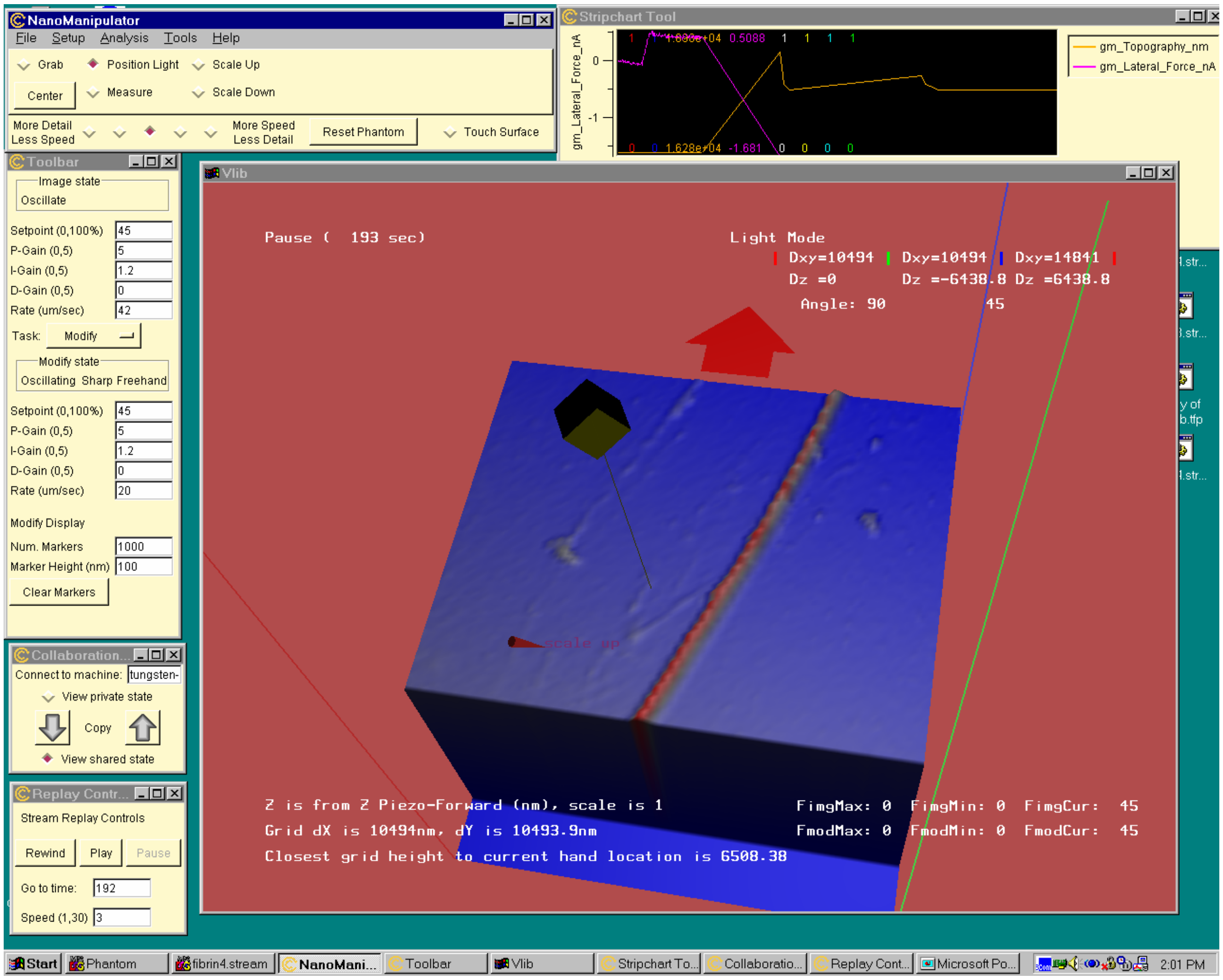


Image state: Oscillate

Setpoint (0,100%)	45
P-Gain (0,5)	5
I-Gain (0,5)	1.2
D-Gain (0,5)	0
Rate (um/sec)	42

Task: Modify

Modify state: Oscillating Sharp Freehand

Setpoint (0,100%)	45
P-Gain (0,5)	5
I-Gain (0,5)	1.2
D-Gain (0,5)	0
Rate (um/sec)	20

Modify Display

Num. Markers	1000
Marker Height (nm)	100

Connect to machine: tungsten-

Go to time: 192

Speed (1,30) 3

```
Pause (  193 sec)                          Light Mode
                                          Dxy=10494  Dxy=10494  Dxy=14841
                                          Dz =0      Dz =-6438.8 Dz =6438.8
                                           Angle: 90          45

Z is from Z Piezo-Forward (nm), scale is 1       FimgMax: 0  FimgMin: 0  FimgCur:  45
Grid dX is 10494nm, dY is 10493.9nm             FmodMax: 0  FmodMin: 0  FmodCur:  45
Closest grid height to current hand location is 6508.38
```

Figure 2. The user interface of the nanoManipulator.

Collaboration System Requirements. The first priority in the collaborative nanoManipulator system is to support the shared cognitive work of collaborative scientific research. This was accomplished by adding functionality to the base single-user nM application and updating its user interface. The second priority was that the system must allow collaborators to communicate and must enable sufficient information transfer between sites that the users are able to develop and maintain a common understanding of the "world" of their collaboration (Sonnenwald, Bergquist et al. 2001; Sonnenwald, Maglaughlin et al. to appear). We addressed the second requirement using off-the-shelf collaboration support products to enable visual and audio communication between the users and to allow sharing analysis tools and other productivity applications.

The Collaborative nM System. In the collaborative system (Figure 3), one PC runs the shared nM application with its attached Phantom™ force-feedback device; the second PC runs Microsoft NetMeeting™ for video conferencing and shared application support. We used a telephone with a wireless headset to ensure high quality, full duplex audio communication. A drawing tablet was available, but seldom used by our study population. We found that the scientists like to have two video cameras; one camera provides a "talking head" view and the other is mounted on a gooseneck so that it can be aimed at whatever the user chooses—typically lab notebooks or apparatus. The local user selects which video signal is broadcast to the remote site.

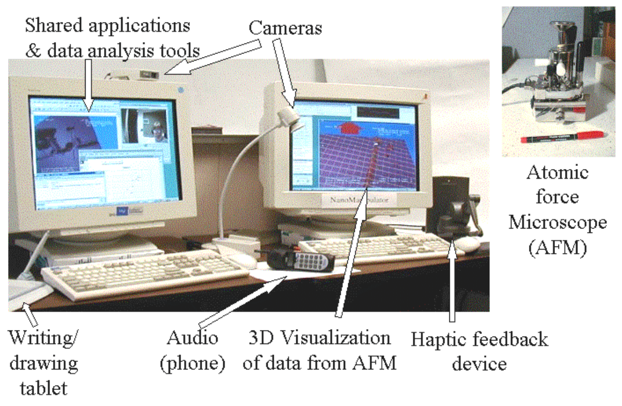

Shared applications & data analysis tools
Cameras
Atomic force Microscope (AFM)
Writing/ drawing tablet
Audio (phone)
3D Visualization of data from AFM
Haptic feedback device

Figure 3. The Collaborative nanoManipulator System.

The shared nM application. The shared nM application is the most important element of the system as it is the users' primary focus when performing their scientific work. We devoted most of our resources to its development. Two specific features were added to the system to support the local user's awareness of the remote user: the remote user's cursor is displayed in the nM window so the local user can see what the remote user is pointing at or attending to. While the local user's cursor changes shape based on active work mode, the remote user's cursor is a constant shape, the red cone seen in Figure 2, and it carries a text tag that indicates mode.

Our observations of the scientists' work patterns revealed that the system needed to support both working individually and working collaboratively (private and shared work modes), and needed to be able to switch back and forth easily between those two modes, including easily moving the results of private work into the shared mode. The widget that lets the user select whether to work privately or collaboratively also appears near the lower left corner of Figure 2.

A Possible Usage Scenario. Two scientists geographically remote from one another can connect to the same AFM using the Collaborative nM. They receive the same data from the AFM and they can select the option to link their visualization and view parameters so that they see the data in the same way (shared work mode). They can discuss the experiment in progress and/or review a past experiment with a common frame of reference that is established via the shared view of the scientific data and the supporting video and audio conferencing tools. Since the visualization-view parameters are kept consistent with an optimistic technique, the users needn't explicitly pass control of the application back and forth, and can, in fact, both be changing viewing parameters at the same time—for instance one user can adjust the orientation of the data plane while the other changes the color map. Either user can break away from the shared viewing mode into the private work mode to explore an idea on his own; then he can easily transfer the results of that exploration back into the shared viewing mode. The system lets the users point to a region of the sample being displayed and be confident that their collaborator sees the same thing they do. Shared analysis tools allow them to use and demonstrate data analysis techniques to each other.

Network Requirements. The network requirements of the shared nM application are listed below (Table 1). Clearly, the bandwidth demands of the system are well within broadband capacities.

Operations requiring more than 100kbps are rare; peak bandwidth consumed during a typical experiment is closer to 20kbps ("Manual Point Data" mode, below).

Table 1 Network requirements of the shared nanoManipulator application.

Data Stream	Bandwidth (bps)	Message Rate (per second)	Observed Latency Tolerance
Microscope Control	6,080	10	> 1 second
Manual Microscope Control	10.880	20	200 ms
Scan Data	20,672	1	> 1 second
Point Data	192,000	250	> 1 second
Manual Point Data	15,360	20	200 ms
Collaboration Data	7,680	10	300 ms

4. Shared nM Architecture

Based on the need for low latency to meet our application's interactivity requirement, our first design decision was to build our distributed system using a fully replicated architecture. A complete copy of the system state was stored at every user's computer, minimizing the time required for any user's program (any *client*) to *read* any part of the state. Replication is useful for minimizing read time, but does not normally speed up *write* operations, and adds considerable complexity to implementations (see Section 2.2). To structure our application replicas, we applied the Model-View-Controller paradigm.

The Model-View-Controller Paradigm

The Model-View-Controller (MVC) paradigm was first introduced as the standard architecture for applications in the Smalltalk programming language and has since been widely used to build many types of interactive applications, including collaborative tools ("groupware"). MVC specifies two facets of an architecture: the division of function among modules and the pattern of communication between modules. Application data and logic are grouped together in the Model, interaction functions into the Controller, and presentation into the View. The Controller instructs the Model to change its state in response to user input (Model state may also change due to internal activities, like simulation). The Model notifies both View and Controller (its "dependents") if any of its data changes, and they are responsible for determining whether or not this change is relevant to them; if the change is relevant to them, they request details from the Model (Figure 4) (Krasner and Pope 1988).

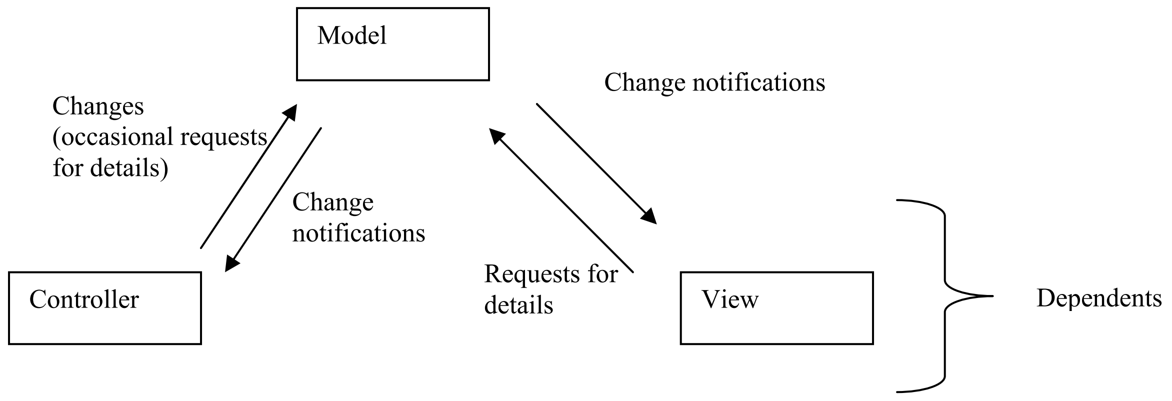
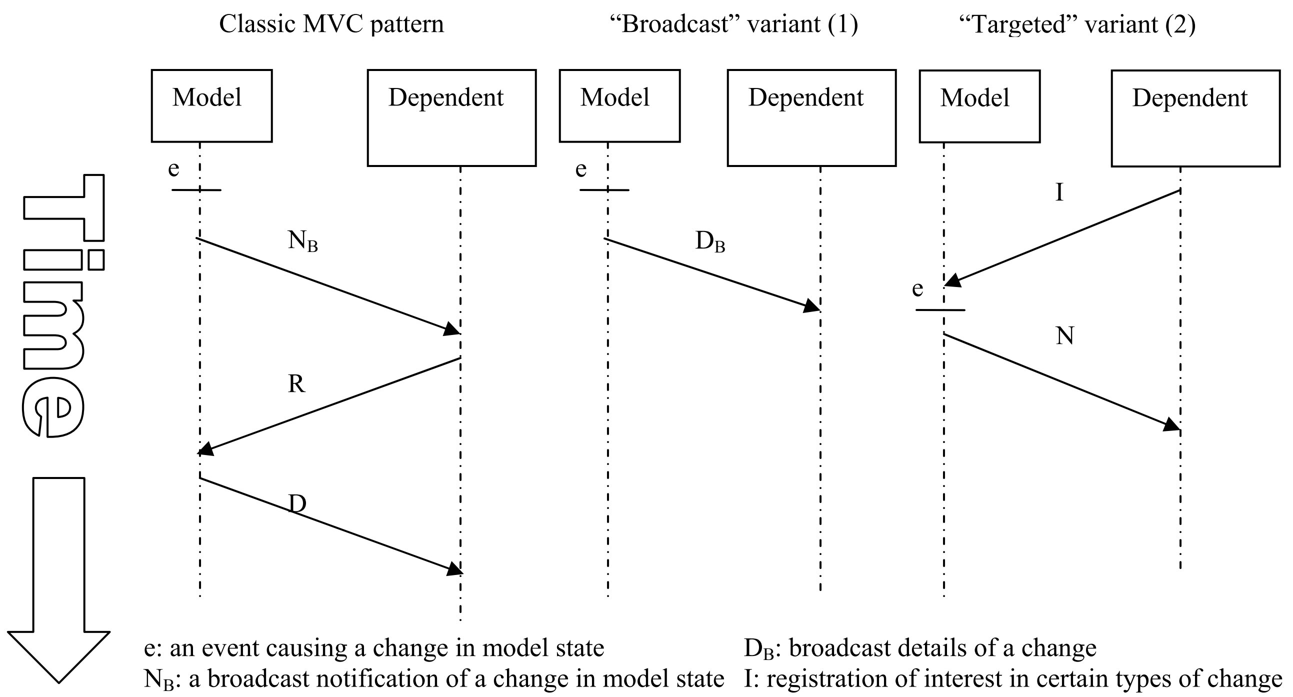


Figure 4. Model-View-Controller components and communication pattern.

There are several reasons for this decomposition. This formalized architecture provides a clear separation of concerns between application logic and application interface, while allowing the interface (both View and Controller) to be specialized to the Model as necessary. Where specialization is not necessary, it promotes reuse. For a modern example, Java Swing user interface components (JFC 2003) are based on a modified MVC pattern to increase their breadth of applicability.

Unfortunately, this standard pattern of communication comes at a price. A cycle of notify-request details-respond is cheap when both components involved are part of the same process, and usually reasonable when they are in separate processes and hosted on the same computer, but the several messages required do not work well when the components are part of a distributed interactive system. The three messages required add one network round-trip-time to the system's latency over any solution that only uses a single message.

This MVC communication pattern has been extended and optimized over time. Latency is reduced either by (1) the Model broadcasting all changes to all dependents, or by (2) dependents registering their interest in specific types of data with the Model, which then sends to each dependent only the updates it is interested in (Figure 5). The first option increases the bandwidth required by the system, while the second increases the complexity of implementing and using the Model, but these tradeoffs are widely accepted. Today, software built "using the MVC paradigm" has a tripartite architectural decomposition but usually one of these new communication patterns instead of the original. It is in this more general sense that we refer to our software as following a Model-View-Controller design. For clarity, Model, View, and Controller are capitalized in the following discussion when the words refer to these components of our software.

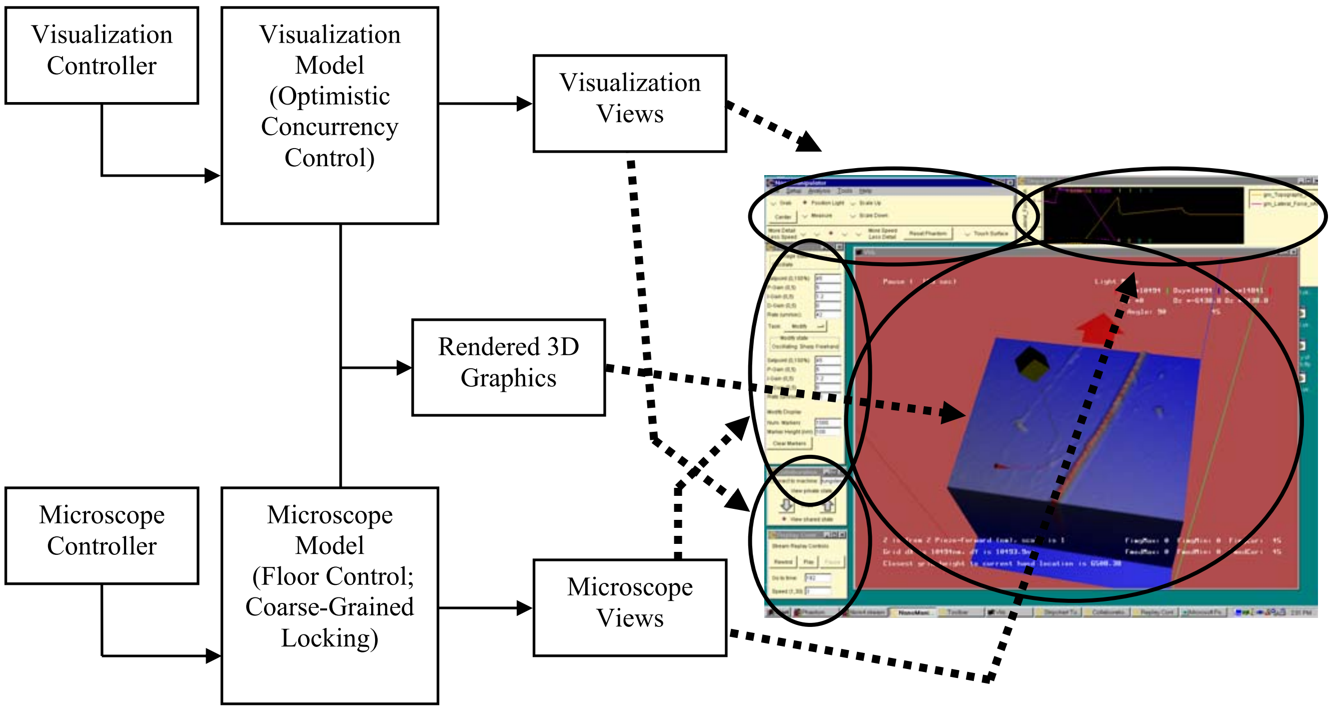


e: an event causing a change in model state
N_B: a broadcast notification of a change in model state
R: a request for details of a change
D: the details of a change
D_B: broadcast details of a change
I: registration of interest in certain types of change
N: a (unicast) notification of a change in model state

Figure 5. Classic Model-View-Controller and two variants.

VEs deal with many different kinds of data: geometry of objects in the world (usually static or slow-changing), position of objects in the world (usually highly dynamic for some subset of objects), rendering modes of objects, and application-specific object state. Most existing CVEs choose one concurrency control method and apply it to all data. We believe this uniformity is unnecessary; these different kinds of data typically have different consistency requirements.

In designing a CVE that uses multiple concurrency control methods, we have found it useful to think of the application as not having a single instance of Model, View, and Controller, but as having several Models, whether in parallel or in a hierarchy, and several Views and Controllers, not necessarily one of each per Model. Each Model can have its own concurrency control method.



Visualization Controller
Visualization Model (Optimistic Concurrency Control)
Visualization Views
Rendered 3D Graphics
Microscope Controller
Microscope Model (Floor Control; Coarse-Grained Locking)
Microscope Views

Figure 6. Two model-view-controller triples capture the different semantics of information (1) about the microscope and (2) about how the users wish to view that information, while the 3D rendering draws from both models to provide an additional view.

Figure 6 shows our decomposition of the CnM into two controllers, two models, and three views. The two Controllers are each integrated with one View as a traditional GUI (implemented using Tcl/Tk) with buttons, sliders, drop-down menus, and other standard widgets to allow users to view and request changes to the current state of the microscope and the visualization methods applied to that state. The third View renders data from the microscope Model using the methods and parameters specified by the visualization Model.

For example, in the Collaborative nanoManipulator system, there is a replicated Model of the microscope. This Model contains such information as, "What is the current location of the microscope tip? What is the current force exerted by the microscope on the sample? What is the topography of the surface recently scanned?" This is the current position and geometry of objects in the world – data sensed by a device external to the virtual world, rather than directly controlled by the user – and the parameters that control that device. The mechanics of Atomic Force Microscopy are discussed in Section 3. A locking mechanism is required for this model because two users trying to direct the tip of the microscope at the same time while it interacts with a sample could cause damage to both microscope and sample and would guarantee that neither user's intention would be satisfied.

The way the data in the Model of the microscope is visualized (View) is controlled by a complex set of parameters, which form a second, subsidiary Model that we call the visualization Model. The data in the visualization Model answer such questions as, "From what position (in a virtual space) is the user viewing the sample? What color map is applied to the sample? How is it illuminated? Are isocontour lines displayed?" This is information about the rendering modes used by the program and about the position and geometry of purely virtual objects.

Control for the visualization Model is very latency-sensitive (from a human-factors standpoint). The operations in the visualization Control are easily commutable and non-critical. The latency requirement and low probability of causing an inappropriate application response suggest that the visualization Model can be treated with optimistic concurrency control.

The data in the visualization-Model defines the difference between shared and private work: if two collaborators have consistent visualization-Models, they see the microscope data in the same way; if not, they work independently. Separating a set of requirements into distinct Models this way, along the borders between feasible concurrency control schemes, helps us determine the architecture of the overall program and helps us build in flexibility.

We can also use this analysis to understand design decisions made in NPSNET, a system for immersive military combat training (Macedonia, Brutzman et al. 1995). Although NPSNET is very different from the nanoManipulator, it similarly shows different consistency requirements for different operations. Like the CnM, NPSNET is a replicated system. NPSNET tracks the movement of players approximately, using low-bandwidth, optimistic techniques to keep the model at the player's local replica updated, with a recovery or convergence algorithm to handle any inconsistency due to optimism (Singhal and Zyda 1999). When a player is shot at, the simulation requires that all players agree on the result of the shot, which can't be done independently if players only know one another's locations independently; this is the sort of operation that must be correct and can not be undone (without seriously violating the Principle of No Surprises). Thus, the determination of whether or not a shell hits is made at one replica and broadcast to all other interested participants. We can think of NPSNET as dividing its world data into two peer Models, each with an appropriate concurrency control method. Player position is kept approximately consistent using optimistic protocols, while player liveness is kept absolutely consistent using centralized decision-making. Like the nanoManipulator, NPSNET uses optimism for speed but centralization for operations that cannot be undone.

Implementing Hierarchical Models

For its visualization-Model, the nanoManipulator uses a fine-grained object-oriented design. For each primitive value (integer, float, or string) in a Model, there is one heavyweight object, which we will call a Variable (subclassed into IntegerVariable, FloatVariable, and StringVariable). The Variable object knows how to display its value in the two-dimensional graphical user interface and accept updates to its value from the user interface. It also implements a callback system, which propagate changes of the Variable's value to all interested entities. (In the language of (Gamma, Helm et al. 1995), this callback system is the *push* variant of the OBSERVER pattern.)

We subclassed Variable, adding two buffers: one for each mode (private or shared). All concurrency control and networking concerns were handled by these buffers, so that we only needed to add to the interface of the primitive value objects the ability to select which mode was active for that object. This separation of concerns was very valuable in debugging and maintenance. When the user changes their collaboration mode, the value in an object's buffer for that mode becomes the object's value, triggering the object's associated callbacks. (These object buffers are similar to the MEMENTO pattern.)

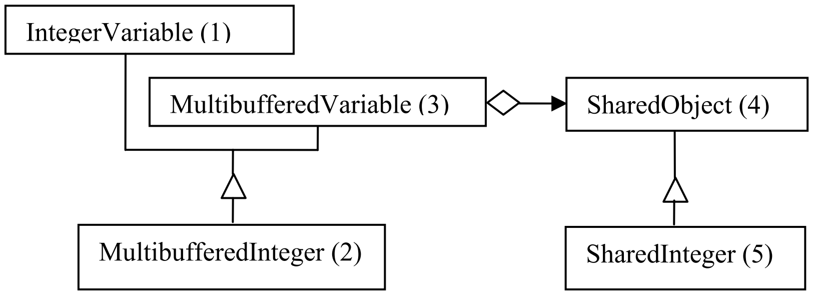


Figure 7. UML diagram of our implementation.

An example is shown in Figure 7. Instances of the preexisting IntegerVariable class (1) were replaced with MultibufferedInteger (3), which inherited both from IntegerVariable and a new utility class named MultibufferedVariable (2). MultibufferedVariable abstracted out the type-independent requirements of managing multiple buffers and switching between them. The buffers were subclasses of SharedObject (4); for a MultibufferedInteger, these were instantiated as SharedIntegers (5). SharedObject and its descendants are classes that know how to keep one value synchronized over a network connection; which value is actually used at any point in time is controlled by the Multibuffered containers.

As defined, this architecture can support arbitrary numbers of users, each with their own private state and all having access to one shared state. If this system were to support additional shared states, each MultibufferedVariable would keep track of more than two SharedObjects. Variables that have special widgets in the GUI are supported by classes inheriting from MultibufferedVariable. To support datatypes other than the standard integer, float, and string, one would add additional pairs of classes descending from MultibufferedVariable and from SharedObject. (The networked SharedObject implementation uses the *push, aspect* variant of the OBSERVER pattern.)

The original nanoManipulator kept Variables as an unstructured "sea" of globals. We converted this into a hierarchy. Using the hierarchy to group related interface parameters together increased the maintainability of the code. It also let us present meaningful chunks of the user interface coherently to the users. For example, all variables related to the color used to render the microscope data – the name of the dataset to map to color, the colormap to apply – are grouped into a "color" submodel. This is grouped with a contour map submodel, lighting submodel, and others into the "viewing parameter" submodel, which, together with other high-level application state, forms the visualization Model (Figure 8).

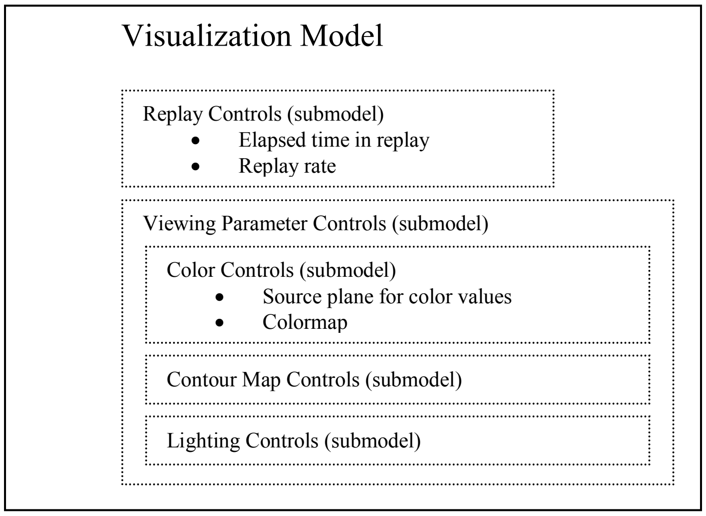

Visualization Model
Replay Controls (submodel)
Elapsed time in replay
Replay rate
Viewing Parameter Controls (submodel)
Color Controls (submodel)
Source plane for color values
Colormap
Contour Map Controls (submodel)
Lighting Controls (submodel)

Figure 8. Some of the hierarchy of objects in the visualization Model.

We have experimented with using this hierarchy to allow the user to mix together their shared and private states, so that they can choose to have some parameters of their visualization View matching a collaborator while keeping others independent. For example, if one user normally labels their data with a colormap in shades of red and green, and another user is red-green colorblind, the colorblind user could apply their own colormap while otherwise seeing everything as the first user specified. In practice, we have found little demand among our users for this kind of mixed View.

The fine-grained approach to maintaining the nanoManipulator's state allowed extensive use of a few base classes and was an excellent design for the original single-user application. When extended to a distributed application, it showed several shortcomings. There are a number of complex objects composed from these primitive value objects; for example, the rendered orientation of the data received from the microscope is a quaternion but is implemented as four independent floating-point numbers. Changes to several of the values comprising a single complex object happened in a coordinated manner within a single process, but when transmitted across the network occurred as distinct changes to the remote copy of the complex object. This caused both unnecessary network traffic and a good deal of difficulty with the design of the callbacks controlling these complex objects. For example, quaternions could transiently take on non-normalized values, which should not be displayed to the user.

A system that would support composition of primitive value objects into complex objects would provide cleaner sharing. This coarser-grained system would not invalidate any of our results or proposed architecture. However, it could violate the separation of concerns (the layering) that placed basic synchronization in the SharedObject and selection in the MultibufferedVariable, since the network synchronization would need to know about the embedding of the SharedObject into the coarse-grained MultibufferedVariable to synchronize each coarse-grained variable atomically.

Programmers have also resisted a coarse-grained system because they perceive it to entail a loss of flexibility. Instead of declaring ad-hoc variables as each is required, programmers using a coarse-grained system must explicitly write code for these clusters of variables to be recognized and treated as a single unit by the networking layer. Sun RPC and Java RMI have solutions to this: data structures or function calls can be analyzed by a preprocess that automatically constructs the code for their network transmission. Similarly, we have built a set of Perl scripts that parses a message-parameter-definition file and automatically writes C++ functions to pack and unpack those data structures (convert them from machine-native representation as several variables to network representation as a string of bytes, and vice-versa).

Synchronizing Multiple Models

We used optimistic concurrency control to keep consistent replicated state in the visualization-Models (Table 2). Although optimistic protocols are intended to provide fast concurrency control, their speed is dependent on the speed of the serialization algorithm they use to agree on the order in which events happen at the nodes of the distributed system. As did NPSNET and AVIARY, we found that wallclock serialization (Hudson 2001) works extremely well for shared VE.

Table 2. Synchronization techniques appropriate for various scenarios.

Technique	Suitable Scenarios
Coarse-Grained Locks	Objects to be acted upon are interdependent, or are considered to be interdependent by users constructing plans
Fine-Grained Locks	Objects to be acted upon can be separated into independent subsets
Explicit Locks	Not losing work is critical, even at the cost of latency and a changed workflow
Implicit Locks	Not interrupting natural workflow is critical, even at the cost of latency or lost work
Optimism	As fine-grained, implicit locks; ideally, conflicting user actions can be automatically reconciled; little risk of damage, easy to undo

We used the Virtual Reality Peripheral Network protocol (Taylor, Hudson et al. 2001)as the Session layer for the Collaborative nanoManipulator, giving us the ability to make asynchronous remote procedure calls (RPC). With asynchronous RPC, when a process requests an operation from another process, the requesting process does not wait for values to be sent back by the remote server, but instead resumes execution immediately. Receipt of the response from the remote server will trigger a callback in the requesting process. This non-blocking communication helps to decouple the application from network latency, allowing an interactive application to reduce response time for operations that are being carried out across the network. In this way, asynchronous RPC is more useful for interactive applications than is traditional synchronous RPC. Asynchronous RPC typically leads to a complex programming style centered on callbacks; however, this same style is also required by modern graphical user interfaces, and was adopted for most of the Collaborative nanoManipulator.

5. Evaluation

5.1 Concurrency Control Performance

The Collaborative nanoManipulator was originally deployed on dual-processor 500 MHz Windows NT workstations machines on a 100 MB switched Ethernet LAN with an Intense3D Wildcat 4000 graphics card. The application was distributed using a fully replicated, peer-to-peer topology. While we expected concurrency control to be a problem when we extended the system across the Internet, we were surprised to find that even on a single campus the latency incurred from simple implementations of non-optimistic or server-based concurrency control was prohibitive.

Deployed on the hardware outlined above, a single user of the system saw a 214 ms response time. Our original collaboration implementation used a server to serialize every operation on the visualization-View parameters, guaranteeing consistency. An arbitrary instance of the application was selected as the serialization server. The user at the server experienced essentially single-user response time, since there was no need to wait for a network round-trip to serialize their actions. However, the remote collaborator saw 319 ms of *additional* latency for every manipulation. Added to the system's native 214 ms response time, this gave a total response time for remote users of over 500ms. Remote collaborators strongly objected to the system's responsiveness, refusing to use the collaborative system. We considered several methods of reducing this latency to a user-acceptable level.

The server-based serialization of the original implementation was slowed by the fact that it was running within the nanoManipulator process, limited by that process' approximately 70 ms frame time. Possible techniques for improving this response time included multi-threading the application or running the serialization server as an independent process. Another latency reduction approach would be to target sources of latency included in the 214 ms that aren't part of the serialization task. For instance, running the application on systems with more modern graphics hardware, e.g., nVidia Quadro Pro 2 graphics, reduces the application's frame time to below 30 ms, which should in turn reduce response time below 100 ms. However, any of these approaches — multithreading, a dedicated server, or newer hardware — would still add an unacceptable 105 ms of network-related latency to the application's base response time.

Another option is to use implicit fine-grained migrating serialization, so that the latency penalty only needs to be paid once for each extended series of manipulations. When a user tries to manipulate one particular object or variable, responsibility for serializing operations on that user's object migrates to the user's host. The initial operation incurs network latency while waiting for serialization to migrate, but once the responsibility has been assigned to the user's host computer further operations do not see any network latency (until another user attempts to manipulate that object and the responsibility is migrated to the other user's computer). However, this is a complex approach that does not completely eliminate serialization latency and does not degrade gracefully. Two users attempting to manipulate the same facet of the application state can experience both surprise (as they repeatedly undo one another's work) and significantly heightened latency (if the responsibility for serialization ping-pongs back and forth between their clients, they pay the serialization latency penalty many times over; less pathological failure modes still leave one user experiencing network latency in their attempts to carry out an operation.)

Our final solution to the problem of high serialization-induced latency was to use wall-clock serialization. Instead of implementing a complex, distributed serialization algorithm, we implemented a method that allows the serialization operation to occur at each site independently. Visualization-View parameter changes that are broadcast to remote sites include a timestamp. The local instance of the shared nM compares the timestamp of the incoming data to the timestamp of the current parameter value and either accepts it or discards it, based on whether its time stamp is later or earlier than that of the current parameter.

We installed off-the-shelf software that keeps the clocks on all of our computers synchronized to the US government's official atomic clock. We observed that the clocks typically stayed synchronized with one another to within 10 ms. Since there is negligible overhead from serialization, this technique gave us performance similar to single-user mode and satisfied our users (Table 3). In practice, even large errors in clock synchronization, on the order of 50-100 ms, do not lead to a visibly inconsistent ordering of events. While response time for our remote users does not achieve our target of under 250 ms, the system is sufficiently interactive to satisfy our users.

Table 3. Performance of concurrency control.

Mode	Mean Response Time (ms)	Mean Response Time due to Concurrency Control (ms)
Single-user (baseline)	214	--
Server-based serialization	533	319
Wall-clock optimism	262	48

5.2 Collaboration System Evaluation

We conducted a controlled experiment to evaluate users' perceptions of the system and their success using it. In the experiment, 20 pairs of upper-level undergraduate science majors participated in two lab sessions during which they used the replay feature of the CnM to explore and analyze AFM data collected earlier. During one lab session the study participants worked face-to-face using the nanoManipulator in single-user mode and during the other lab session they worked remotely (in two different locations) using the Collaborative nanoManipulator. Order of the two conditions, face-to-face and remote, was counterbalanced, i.e., ten pairs worked remotely first and ten pairs worked face-to-face first. A detailed description and discussion of the experiment can be found in (Sonnenwald, Whitton et al. 2002). In this section we focus only on interview data related to concurrency control and, specifically, user comments comparing the explicit, coarse-grained floor control of the analysis and productivity applications and the implicit, optimistic concurrency technique employed for shared visualization-view parameters.

After each lab session, we interviewed participants asking for their perceptions of the system. In particular, they were asked what they found most satisfying and dissatisfying about using the technology. The questions were open-ended and participants were free to discuss any aspect of the technology, including visualization functionality, scientific data analysis tools, haptic feedback, audio and video conference capabilities as well as concurrency control. Eight participants (out of 40) specifically discussed the relationship of concurrency control and usability of the system in their responses. The responses indicate that participants preferred the ability to work simultaneously in the shared nM application, which uses optimistic concurrency control, over the explicit floor control required in the off-the-shelf shared application software, Microsoft NetMeeting. Participants commented:

> I think…the best feature…was the ability to work on the same thing at the same time with the nanoManipulator…to work on the same image at the same time doing different things…was key…The thing that frustrated me the most was the shared applications…you could see the other person doing things but you couldn’t do anything…On the nanoManipulator, I could position one measurement cursor while [my partner] positioned the other measurement cursor, and that was really nice…You can’t do that kind of stuff in the shared applications…[This was] frustrating to me.

> It was nice to be able to share stuff because if I didn’t know something or I didn’t ant to do [something], [my partner] could do it. that [sharing] was really easy. The aggravating part [of shared applications] was that you had to fight over the mouse: ‘I want to type; click, click, click’ or ‘No, I want to type’, and having to go back and forth…It’s like little kids smacking each other…We never fought over the nanoManipulator because it was more interactive as far as both of us being able to use it at the same time.

> The [application] sharing thing is hard because it’s hard to designate who’s in control…sometimes…you want to fix something but you realize you’re not in control and you’re like ‘Wait, wait a minute’…it’s confusing…[With respect to sharing on the nanoManipulator], it’s like you each have your own control and you can do things…I liked the sharing with the nano.

Participants used verbal communication between themselves to help mediate the issue of control. As one participant explained:

> The shared applications…became exceedingly frustrating...For example, when I wanted to do something, my partner wanted to do something at the same time, and several times we went back and forth double-clicking to gain control… Essentially…we were fighting over control…the audio went a long way to help out in that regard. We could communicate and say ‘Hey, I’m going to do this’ or ‘As soon as I’m done with that then you go ahead and take control.’

Some participants also preferred the Collaborative nanoManipulator system over working face-to-face because they each had full access to a system, they could go into the private work mode and make progress independent of their lab partner, they could cooperate to accomplish a task, and/or they could work in parallel to accomplish the entire task more quickly. Participants explained:

> [When working remotely} I liked that we each had our own [system] and we can choose whether we were going to share mode or not…but still you can easily…say: ’here, can you just go into grab mode and move it for me?’ you’re not constantly [having to ask] ‘OK, you sit here and you do it.’

> I liked that we were [physically] separate…If one of us got snagged up with something, the other could independently work and get it done rather than both of us being bogged down by having to work on it simultaneously. We could also multi-task; one of us could work on Part A, the other on Part B, and so on. [That] change of pace was the most satisfying thing about [the lab.]

In summary, collaborating study participants reported that the optimistic concurrency control as implemented in the shared nanoManipulator application provided advantages over the explicit floor control found in both the shared applications and in face-to-face cooperation. Verbal communication was used to help resolve conflicts that arose when control had to be explicitly passed. In comparison, participants did not report extra verbal communication was needed to mediate sharing in the shared nM application.

6. Conclusions and Future Work

Building the Collaborative nanoManipulator, we found that the Model-View-Controller paradigm was a useful way to analyze CVEs. Extending Model-View-Controller to include multiple parallel or hierarchical Models highlights the differing concurrency control requirements of different subsets of an application's state data and helps us design the application's architecture.

To support the interactivity requirements of our CVE, we used asynchronous RPC and optimistic concurrency control. Optimism is a valuable technique particularly well suited for minimizing latency of continuous, easily reversed inputs to a VE, such as a users' viewpoint position and orientation. It is less amenable to supporting transactions and complex assemblies of primitive operations, but can be used to do so with a moderate software engineering effort.

Our research was performed in the context of replicated, distributed, collaborative virtual environment-based applications with small numbers of users. In general, application replication scales poorly as a technique for supporting many users in interactive applications. DVEs with hundreds and thousands of users require more complex concurrency support than that used in our system, typically including area-of-interest management and the reintroduction of servers to filter communication (Macedonia, Brutzman et al. 1995). It is our belief that our analysis and implementation techniques are applicable even in systems that have these additional layers in their architectures.

All of these techniques – from high-level analysis and user interface design down to details of implementation – should prove helpful for other groups implementing a wide variety of CVE systems.

Acknowledgements

More than a dozen computer and information scientists worked on the nanoManipulator project during the development and initial deployment of the system; in particular, Kelly Maglaughlin, Eileen Kupstas Soo, and Ron Bergquist contributed to the development of the user requirements. Russell Taylor and several anonymous reviewers had suggestions that strengthened the original paper, presented at IEEE VR2003. We must also thank the many physicists, chemists, and biologists who have worked with the nanoManipulator project, and the NIH National Center for Research Resources (Grant P41 RR 02170) and the National Institute for Biomedical Imaging and Bioengineering (Grant P41 EB 002025).

Bibliography

(JFC 2003). Java Foundataion Classes (JFC)/Swing, Sun Microsystems, Inc. http://java.sun.com/products/jfc/

(NCMIR 2003). National Center for Microscopy and Imaging Research. http://ncmir.ucsd.edu/Telescience/

Berman, H. M., J. Westbrook, et al. (2000). "The Protein Data Bank." Nucleic Acids Research **28**: 235 -- 242.

Bhola, S., G. Banavar, et al. (1998). Responsiveness and Consistency Tradeoffs in Interactive Groupware. CSCW.

Bryson, S. and S. Johan (1996). Time Management, Simultaneity and Time-Critical Computation in Interactive Unsteady Visualization Environments. Proceedings of IEEE Visualization.

Gamma, E., R. Helm, et al. (1995). Design Patterns: Elements of Reusable Object-Oriented Software. Reading, MA, Addison-Wesley.

Herlihy, M. (1987). "Concurrency versus Availability: Atomicity Mechanisms for Replicated Data." Transactions on Computer Systems **5**(3): 249 - 274.

Herlihy, M. (1990). "Apologizing Versus Asking Permission: Optimistic Concurrency Control for Abstract Data Types." Transactions on Database Systems **15**(1): 96 - 124.

Hudson, T. C. (2001). Concurrency Control for Collaborative 3D Graphics Applications. Chapel Hill, University of North Carolina.

Krasner, G. E. and S. T. Pope (1988). "A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80." JOOP: 26 - 49.

Leckenby, J. (2000). A Practical Guide to Scanning Probe Microscopy, ThermoMicroscopes.

Macedonia, M. R., D. P. Brutzman, et al. (1995). NPSNET: A multi-player 3D virtual environment over the internet. I3D, ACM.

Macedonia, M. R., M. J. Zyda, et al. (1994). “NPSNET: A Network Software Architecture for Large-Scale Virtual Environments.” Presence **3**(4): 265 - 287.

Malpani, R. and L. A. Rowe (1997). Floor Control for Large-Scale MBone Seminars. ACM Multimedia, Seattle, WA, ACM.

Meehan, M. (1999). Survey of Multi-user Distributed Virtual Environments.

Olson, G. M., D. E. Atkins, et al. (1998). The Upper Atmospheric Research Collaboratory. interactions**:** 48 - 54.

Potter, C. S., B. Carragher, et al. (2000). Bugscope: A Practical Approach to Providing Remote Microscopy for Science Education Outreach. Urbana, IL, Beckman Institute, UIUC**:** 10.

Ressel, M., D. Nitsche-Ruhland, et al. (1996). An Integrating, Transformation-Oriented Approach to Concurrency Control and Undo in Group Editors. CSCW, Cambridge, MA, ACM.

Singhal, S. and M. J. Zyda (1999). Networked Virtual Environments: design and implementation, ACM Press.

Snowdon, D. and A. West (1994). “Aviary: Design Issues for Future Large-Scale Virtual Environments.” Presence **3**(4): 288-308.

Sonnenwald, D. H., R. E. Bergquist, et al. (2001). Designing to Support Collaborative Scientific Research Accross Distances: The nanoManipulator Environment. Collaborative Virtual Environments. E. F. Churchill, D. N. Snowdon and A. J. Munro. London, Springer-Verlag**:** 202 - 224.

Sonnenwald, D. H., K. L. Maglaughlin, et al. (to appear). “Designing to Support Situation Awareness across Distances: An Example from a Scientific Collaboratory.” Information Processing & Management.

Sonnenwald, D. H., M. C. Whitton, et al. (2002). “Evaluating a scientific collaboratory: Results of a controlled experiment.” ACM Transactions on Computer Human Interaction **10**(2): 151--176.

Taylor, I., Russell M, T. C. Hudson, et al. (2001). The Virtual-Reality Peripheral Network (VRPN) System. Chapel Hill, NC, University of North Carolina at Chapel Hill**:** 10.

Taylor, I., Russell M and R. Superfine (1999). Advanced Interfaces to Scanning Probe Microscopes. Handbook of Nanostructured Materials and Nanotechnology. H. S. Nalwa. New York, Academic Press**:** 271 - 308.

Taylor, V. E., J. Chen, et al. "Immersive Visualization of Supercomputer Applications: A Survey of Lag Models.".

Vaghi, I., C. Greenhalgh, et al. (1999). Coping with Inconsistency due to Network Delays in Collaborative Virtual Environments. VRST 99, London, UK, ACM.

Ware, C. and R. Ralakrishnan (1994). "Reaching for Objects in VR Displays: Lag and Frame Rate." Transactions on Computer-Human Interaction **1**(4): 331 - 356.

Weihl, W. E. (1993). Transaction-Processing Techniques. Distributed Systems. S. Mullender, ACM Press**:** 329-352.

Wickens, C. D. and P. Baker (1995). Cognitive Issues in Virtual Reality. Virtual Environments and Advanced Interface Design. W. Barfield and I. Furness, Thomas A. New York, Oxford University Press**:** 514 - 541.