# Concurrency Control for Collaborative 3D Graphics Applications

Thomas C. Hudson
University of North Carolina at Chapel Hill, USA
*hudson@cs.unc.edu*

**Abstract.** Interactive 3D graphics applications become unusable with concurrency control mechanisms that impose significant latency. We show how frameworks from the collaboration literature that have been applied to text editors can be extended to collaborative 3D graphics tools. We have implemented several techniques for concurrency control in the DISTRIBUTED NANOMANIPULATOR, a shared 3D interface to Atomic Force Microscopes. We show how the choice of concurrency control mechanism determines our tool's usability.

## Introduction

Improvements in hardware and graphics libraries have allowed interactive 3D graphics applications to become increasingly widespread, but distributing these applications across a network to multiple interactive users is still an open problem. A rule of thumb in graphics and user interface design is that interactive frame rate needs to be at least 15 or 20 Hz — that is, that the system must respond to user input within 50 to 70 ms. Single-user sys-

tems can now meet these latency targets, but network delays by themselves can exceed the entire latency budget, reducing applications to noninteractivity (See appendix). This paper addresses concurrency control, the central problem in effectively sharing interactive 3D graphics applications between simultaneous, widely-separated users.

> A concurrency-control protocol ensures that incorrect behavior cannot occur as a result of concurrent access by multiple clients. [Her87]

Concurrency control attempts to guarantee that all users see a consistent application state. Collaborators editing a document together should see the same text. The details of the consistency guarantee are highly application-dependent.

Concurrency control also attempts to guarantee that users see an expected application state, sometimes expressed as the "Principle of No Surprises", or the "Isolation Property" of databases [Wei93]. Users should never be surprised by the application's response to their input or have unintended effects due to another user's simultaneous actions.

Since distributed databases may be accessed simultaneously by several users, they require concurrency control. The typical approach taken by databases is to lock records being modified so that only one user can operate on a record at a time. A lock is secured on the records needed by the operation, the operation is performed, and then the lock is released. This sequence of steps introduces latency [Wei93].

Optimistic concurrency control is a class of algorithms that guarantee consistency with minimal latency. Following Amdahl's Law — *make the common case fast* — systems using optimistic concurrency control perform an operation, then check to see whether the operation conflicted with other simultaneous operations at different hosts. If conflicts occur, an application-dependent procedure resolves them [Her90]. Most operations do not conflict, so can be executed rapidly, without the overhead of locking.

Latency interferes significantly with interactive applications, particularly those with direct manipulation interfaces. In a 2D mouse-driven drawing application, users notice as little as 50 ms of system latency and react negatively to 80 ms or more. Three-dimensional interaction is also latency-sensitive [WB94]. Because of this sensitivity to latency, many groupware systems use optimistic concurrency control.

Optimistic concurrency control has to be careful to preserve intention. It was introduced for text editors in the dOPT and AdOPTed algorithms [EG89, RNRG96], which provide both consistency and intention-preservation. These algorithms rely on a simple semantics for editing text. Recent work has extended optimistic approaches to spreadsheets and 2D graphics editors.

This paper surveys concurrency control approaches in the context of collaborative 3D graphics applications to build an understanding of which modes of concurrency control are appropriate for any given application. We describe and measure the performance of our implementation of optimism in the DISTRIBUTED NANOMANIPULATOR, a collaborative virtual-reality interface to an Atomic Force Microscope that is a simple case for optimism [SBM+01, HSM+00]. We describe a multiuser Computer-Aided Design (CAD) system where more complex protocols based on dOPT are needed for optimism [Hud99].

## Context

The NANOMANIPULATOR is a 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 [TCO+97]. The program allows scientists to see, feel, and modify samples ranging from DNA and buckytubes to viruses and cells.

We extended the NANOMANIPULATOR into a collaborative application, the DISTRIBUTED NANOMANIPULATOR. Several scientists can connect to the same Atomic Force Microscope. They all receive the same data, and can choose to couple their displays so that they all see the data in the same way. This lets them discuss an experiment in progress or review a past experiment with a common frame of reference, lets them point to a region of the sample under the microscope and be sure that their peers see the same thing they do, and lets them demonstrate data analysis techniques to one another.

# Theoretical Approaches to Concurrency Control

## Replication

One simple approach to concurrency control is to use a single server to hold application state. All clients read from and write to the server; since there is only one copy of the state, there is no inconsistency. However, this means that clients suffer one network round trip time of delay in performing any operation. In order to speed up client operations, state can be replicated at every client's computer.

This paper addresses the problem of concurrency control faced by replicated applications with small numbers of users. Replication scales poorly; hundreds and thousands of users require additional approaches, typically including area-of-interest management and the reintroduction of servers to filter communication [MZP+95].

## Concurrency Control

Concurrency control attempts to preserve the intent of all users of a distributed application while allowing them to work in parallel. Approaches to intention preservation fall into two classes, pessimistic and optimistic. An algorithm also has to make a second independent choice of how to serialize input from all the users to synthesize a consistent state.

Concurrency control needs to insulate users from one anothers' inputs. If two users try to manipulate the same object at the same time, they will see some interleaving of their actions that may not do what either user intended.

Concurrency control is governed by Singhal's Consistency-Throughput Tradeoff:

> It is impossible to allow dynamic shared state to change frequently and guarantee that all hosts simultaneously access identical versions of the state. [SZ99]

### Pessimistic Approaches

Pessimistic approaches try to prevent users from requesting any operations that could conflict. The simplest way to do this is to have a lock.

With an explicit lock, only one user at a time can actively make changes to

the data. Inactive users press a button to request control of the application. This is a classic approach in the Computer-Supported Cooperative Work (CSCW) literature, where it is called "floor control" [Lan86]. As in a well-run meeting or presentation, only one speaker at a time "has the floor" and can address others or take action, while others must watch and listen.

A widely disseminated system that uses explicit locking is Microsoft's NetMeeting. NetMeeting allows users of Windows workstations to share a wide variety of 2D mouse-driven applications. Only one user has control of the on-screen mouse pointer at any time; others watch. If another user clicks his mouse, he takes control — after waiting for network traffic to deactivate the previously active user's mouse. In our studies, users have found this explicit floor control both confusing and slow, interfering with their collaboration. Many collaborative systems require the current active user to explicitly relinquish control before another user can take control; some also support queueing requests for control by inactive users.

Even in large documents, floor control allows only one user to work at a time, removing any possibility of parallel activity. To allow multiple users to be active at once, locks can be made fine-grained. In a text document, each paragraph could have its own lock. In a virtual world, each object could have a lock.

Managing fine-grained locks can be difficult for users, since they may become as complex as the document. To avoid the overhead of management, fine-grained locks are commonly made implicit: the user takes no action outside the normal flow of work to request the lock. As soon as they undertake some manipulation that would require the lock, their local interface blocks their progress until the lock is obtained. Once they have the lock, they are allowed to continue with their operation. If they can not obtain the lock because another user is holding it, their local application may unblock and display a failure indicator, or continue to block until the other user is finished. When the operation is complete, the lock is released.

Computers do not always know when to release a lock. Consider a user in a 2D shape editor who moves an object, releases the mouse to consider the location briefly, and then decides to move it further. If he waits too long between the two movements, the concurrency control system may release his lock. He must then wait for the network round-trip needed to regain the lock, and must also contend with other users who may be trying to

carry out different operations on the same object. This time penalty can be partially ameliorated by migrating a lock to the workstation of the last user who requested it. If user A uses an object, releases it, then uses it again, no network traffic is needed, since during his first use of the object, responsibility for managing the lock was moved to his workstation. If user B wants to use the object, she must send a message to A's workstation rather than to a central lock server. If A and B both try to use the object frequently, the lock will migrate frequently, slowing the system.

The added latency from fine-grained implicit locks comes from blocking from when the user attempts an operation until the lock has been obtained. Some systems allow the user to begin the operation as soon as the lock is requested, but then abort (undo) the operation if the user cannot obtain the necessary lock. This minimizes latency, but causes surprise and lost work when conflicts occur (when two users try to operate on the same object simultaneously).

Optimistic concurrency control protocols carry this approach one step farther: they allow the user to begin working immediately, and take some step less extreme than aborting if they detect that the user's imput conflicts with another user's actions.

### Optimistic Approaches

Optimistic concurrency control avoids the time overhead of locking. Each operation is applied at a user's workstation as soon as she requests it. Her operation is then sent to other workstations, which are responsible for executing it in such a way as to guarantee consistency. Optimism requires that an ordering exist over all operations, but does not require that operations be delivered in order.

The simplest case of optimism occurs when all operations are commutative or idempotent and need not be serialized. Operations on an object arriving that predate the most-recently-executed operation on that object can be discarded or executed out-of-order as appropriate.

To use math as a simple example, the operation "set X to 4" is idempotent. If two users issue this operation, we can execute it twice, or ignore the second instance. If we instead consider only the operations "increment X" and "decrement X" our system is commutative: every operation to increment or decrement can be executed when it is received, and all users will see the same

final state. With these sets of operations, we need not even have an ordering.

However, if we combine "set" with "increment" and "decrement", we need to have a global ordering. "X++; X = 4" does not have the same result as "X = 4; X++". Now, every replica of our system needs to agree on which happened first: this is the problem of serialization. Different approaches to serialization can radically change the amount of time and network traffic required for concurrency control.

Even when a global ordering has been established, the operations "increment" and "set" cannot be reconciled by reordering: the host executes "X++", then receives an operation "X = 4" which the ordering says should precede the increment. The host needs to take one of two approaches, each of which has spawned a body of work in the CSCW community. One approach is to undo "X++", do "X = 4", and then redo "X++". This works well as long as operations can be cheaply undone and operations are only slightly out-of-order. The second approach is to use the semantics of increment and set to transform "X = 4" into "X = 5" and execute the transformed operation. This works well as long as the number of operations is small and their semantics are well-behaved. Both undo/redo and transformational approaches have been used for shared applications that edit text and manipulate 2D graphics.

### Serialization

If operations are just broadcast from every replica to every other replica without additional data, the order in which they occur is unknown. Some additional information is necessary to put them in a consistent order, or *serialize* them. The goal of optimistic concurrency control is low-latency performance. Total latency depends on the serialization technique used.

The simplest way to serialize is to centralize: all message traffic passes through a single server, which is responsible for giving it a consistent ordering. Every replica sends operations requested by its user to the serialization server, which then broadcasts them to all replicas, including the originator, in the order in which it received them; the replicas execute the operation when it is broadcast to them. If operation A arrives at the serialization server before operation B, then every replica in the system executes A before executing B, regardless of the time at which they were requested by users. Centralized serialization defines a total order over all operations in the system, and makes

sure that they are delivered to all replicas in the same order.

The drawback to centralized serialization is that every operation experiences one full network round-trip of latency. Instant response is impossible; a user cannot see the results of her action until it has been to the serialization server and back. Over a wide area network, this delay can be hundreds of miliseconds, enough to interfere with the application's usability [Pra99]. The performance of centralized serialization is similar to that of locking.

The latency penalty can be reduced by distributing the responsibility of serialization. Operations on a single object may only need to have a consistent ordering with respect to that object. In this case, serialization for operations on the object may be migrated to the system that most uses the object. By moving serialization out of a central server and into the endsystems, each object can have one user who sees no latency to operate on it; as the use pattern of an object changes, serialization for that object can move to whichever replica uses it most. However, distributing the serializer like this makes it impossible to serialize operations that involve multiple objects. Migrating, fine-grained serialization has a performance similar to migrating, fine-grained locking schemes, which have been widely proposed as a mechanism for concurrency control in shared virtual reality. Optimistic concurrency control protocols should be able to do better.

Using a logical clock, we can get rid of the need to have any serializing server [Lam78, BM93]. The logical clock algorithm is described briefly in an appendix. Although not dependent on a server, logical clocks do not place a message into a total order until subsequent messages have been received from every peer in the system. To minimize delay in a system using a logical clock requires a large bandwidth flow between all peers.

The easiest way to serialize operations is to tag them with the wall-clock time at which they were requested. Although this has long been made difficult by clock mismatch, it is practical with Network Time Protocol [Mil96] so long as the frequency of operations does not approach the error bound in clock synchronizations. Schneider uses wall-clock synchronization plus an extra delay to provide pessimistic concurrency control [Sch93]. Wall-clock synchronization does not guarantee in-order delivery of messages without this extra delay, but the optimistic intention-preservation methods discussed above do not require in-order message delivery.

## Model – View – Controller

The Model-View-Controller (MVC) paradigm [KP88] was first introduced as the standard design of applications in the Smalltalk programming language, and has since been widely used to build many types of interactive applications, including collaborative tools ("groupware") [GUN96]. It specifies two facets of an architecture: the division of function among modules and the pattern of communication between modules. Application data and logic is grouped together in the Model, input-handling functions into the Controller, and output into the View. The Model notifies both View and Controller if any of its data changes, and they are responsible for determining whether or not this change is relevant to them.

The communication pattern of Model-View-Controller is more suited to implementation on a single machine than it is to being distributed across a network. Every update from a model to a remote view or controller has at least 3 messages cross the network: a notification from the model, a query from the remote module, and then the actual data from the model. Several groups have made extensions or optimizations that improve its performance [SSS99]. The most common approach is to allow the model to push data to the remote modules preemptively.

Scientific 3D graphics applications may have multiple modes or input modalities with differing requirements for concurrency control. These different modes and modalities typically deal with disjoint subsets of the models, and user intent does not "cross over" between them. In designing the DISTRIBUTED NANOMANIPULATOR, it proved useful to think of these applications as having not one model-view-controller, but several models, hierarchical or interconnected, each with its own views and controllers. Each MVC triple can have its own concurrency control method. The view and controller exposed to a user might be the union of all the partial instances, or a subset of them, with other views and controllers entirely internal to the application.

For example, in the DISTRIBUTED NANOMANIPULATOR, there is a replicated model of the microscope whose consistency is guaranteed by sending all updates through the microscope control server for serialization, and where intention is preserved by a coarse, explicit lock. The view of the data in this model of the microscope is controlled by a complex set of parameters, which may be regarded as another model, whose control is much more latency-sensitive (from a human-factors standpoint) but whose operations

are commutable and non-critical. The view-model is treated with optimistic concurrency control.

Thinking of these microscope and view data as separate but connected instances of the Model-View-Controller paradigm helped us design the infrastructure to maintain both consistency and user intent. Instead of treating all of the application's state as a single model, we break it up into two models. One model is the microscope's state and the data the microscope has reported. The controller for this model is replicated at every user, but a locking protocol guarantees that only one user can modify it. The other model is the set of parameters that defines the view of the microscope. The controller for the view parameter model is replicated at every user, and uses optimistic concurrency control.

NPSNET is a contrasting example of user intent crossing over between different modes. NPSNET is a research system for military combat training meant to succeed DISTRIBUTED INTERACTIVE SIMULATION (DIS) [MZP+94]. Different consistency requirements hold for different operations. Consider movement and firing weapons: The position of soldiers and units is displayed approximately to all players, using low-bandwidth, inexact optimistic techniques to keep the model at the player's local replica updated, with a convergence algorithm to handle errors from optimism [SZ99]. When a player is shot at, the system requires a consistent answer that can not be left to the mercy of the normal approximate position data. The "shoot at" operation is executed at the target player's host. This is a good solution to the application's specific consistency constraints, using optimism for speed but switching to centralization for operations — like death — that can not be undone.

Separating a set of requirements into distinct models along the borders between feasible concurrency control schemes both helps us determine the architecture of the program and helps us build in flexibility. The view of the microscope data in the DISTRIBUTED NANOMANIPULATOR is normally shared, but can be uncoupled so that each user has a different view. We accomplish this by uncoupling the view-models; nothing in the "primary" model-view-controller system for the microscope changes.

# Concurrency Control for 3D Graphics

[Mee99] is a survey of multi-user distributed virtual-environment systems that shows the wide range of concurrency control approaches that have been tried. These range from no concurrency control at all [Bro97] to central servers providing a total ordering [MF96, Fun95], from fine-grained per-object locks [BWA96], to a distributed total ordering mechanism [Mee98]. NPSNET and AVIARY use a wallclock ordering approach similar to that we used for the DISTRIBUTED NANOMANIPULATOR [MZP+94, SW94]

The two models in the DISTRIBUTED NANOMANIPULATOR, the microscope and the view parameters, are at two extremal points in the continuum of concurrency management problems: one can only be handled by coarse locking, while the other can use pure optimism and rely on clock ordering to solve conflicts. There are interactive 3D graphics applications that lie in the middle ground between these extremes, where optimism will require a more complex conflict-resolution scheme to preserve both consistency and user intention.

One of the most interesting of these applications is collaborative Computer-Aided Design (CAD). Large, complex objects, like oil rigs or power plants, will often have several engineers working in the same space, each laying out the components of a different system. A typical installation has the engineers design independently all day on their workstations, then uses a large centralized database to merge the day's work at night. Only during this nighttime batch process are conflicts that arose during the day detected. Thus, an engineer can come in one morning to find the previous day's work, depending on a layout decision she made at 8:30 AM, conflicted with a decision one of her peers made at 8:15. If the workstations communicated with a server, or one another, during the day, these conflicts could be detected and resolved much more quickly, reducing rework and wastage.

# Experience with Concurrency Control

## Distributing the nanoManipulator

There are several frameworks and toolkits for building multi-user distributed virtual environments [Mee99]. Working with a large existing application, we

found it easier to write our own support software for concurrency control (10000 lines of source) than to rewrite the entire application (125000 lines of source) to fit within these external frameworks. This gave us considerable freedom to experiment with various concurrency control methods and to tune them to suit the application semantics.

In the shared application, there are two major places where concurrency control is necessary: first in control of the microscope, and second in control of the view parameters when the users have synchronized their views.

Microscope control is one place where concurrency must be avoided: to have even 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 ensure that neither user's intention would be satisfied. If we used fine-grained implicit locking, we could guarantee that continuous commands (e.g. motions of the microscope tip) would not be interleaved, but could not guarantee that sequences of commands would be preserved. In complex systems, optimistic concurrency control often relies on the ability to "undo" an action, and although that works well in a text editor, it works poorly across a physical interface to the real world. The nature of teleoperation means that shared teleoperators will generally have to use locking; to assure safety and preserve user intent, our teleoperation application requires coarse-grained, explicit locking for microscope control.

View parameter control is amenable to concurrent changes by our collaborating users. All operations are commutative or idempotent. Any error in view parameter control can be swiftly fixed; there are no irreversible repercussions. This allows us to reconcile commands without operation transformation. Synchronized clocks are used to order all updates to the view parameter model, and out-of-order updates are discarded.

Our DISTRIBUTED NANOMANIPULATOR is deployed on WINDOWS NT workstations, typically dual-processor 500 MHz machines on a 100 MB switched Ethernet LAN. We expected concurrency control to be a problem when we extended the system across the Internet, but found that even on a single campus the overhead from simple implementations of concurrency control became prohibitive.

Our original implementation used a server to serialize every operation on the view parameters. An arbitrary replica was selected as the serialization server. The user at the server saw good response, since there was no need to

wait for a network round-trip to serialize their actions. However, every other user in the system saw 319 ms of extra latency for every manipulation, enough when combined with the system's 214 ms update time that they objected strongly to the added latency, refusing to use the collaborative system.

A straightforward improvement would be to use fine-grained migrating serialization, so that the latency penalty only needs to be paid once for each manipulation. Users would experience extra latency at the beginning of an operation but no latency thereafter. This initial spike of latency still interferes with use. Without careful tuning, migrating serialization also fails to degrade gracefully: when users interfere with one another's work or try to work closely together in manipulating the same parameters, the system becomes less responsive as serialization ping-pongs back and forth.

We instead used wall-clock serialization, which gave us essentially the same performance as single-user mode (Table 1), and satisfied our users. So long as user input is less frequent than the error in clock synchronization, all users will see a consistent ordering of events.

| 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 |

Table 1: Latency cost of tested concurrency control methods. Response time is time between receipt of user input and display of corresponding output. Mean and peak are computed for every user input during a run of the program.

## Sharing Computer-Aided Design

Collaborative virtual environments (CVE) have been used for engineering design reviews for several years [LJDea99]. Although design reviews allow engineers and clients to inspect completed designs, engineers who are working together need to inspect one another's work on a continual basis to avoid making conflicting design decisions. As we discussed in the preceeding section, modern CAD tools do not support continuous awareness. A typical system accepts every engineer's work during the day and detects collisions

in a nighttime batch process. This is a particularly coarse form of optimism, where conflict results in an entire day's work having to be undone.

We propose to replace the overnight conflict detection with a system that provides the engineers with a continuous low level of awareness of others' work in the same space and gives them immediate feedback when a conflict occurs. To ensure acceptance, this real-time collaborative system needs to be every bit as interactive as current standalone systems. Detecting a conflict is usually a CPU- and memory-intensive task best handed off to a server. Locking, whether implicit or explicit, presents both algorithmic and human-factors problems [RRD90], so optimistic concurrency control is called for. Some preliminary work was done in [Hud99], which discusses transformational approaches to resolving conflicts and automatic versioning when designers are trying to arrange parts in a common 3D space.

## Conclusion

The typical solution to concurrency control in shared 3D graphics applications is locking. Unfortunately, this adds extra latency to human interaction with the system. The additional latency from concurrency control can make the system essentially unusable.

This was the case with our implementation of the DISTRIBUTED NANOMANIPULATOR — locks made the system too slow to use, even when they migrated to the site of the requestor. Lock migration actually makes the system perform worse in situations of high contention. Using optimistic concurrency control gave us good performance, helped by the fact that the portion of our user interface that was most sensitive to latency was also most amenable to optimism and did not require the higher overhead of transformational and undo-based schemes.

Concurrency control methods vary in suitability depending on both the object operated upon and the operation to be carried out. In order to preserve user intention, operations that depend on the current state of the application have more stringent consistency requirements than operations that are not state-dependent. Operations that cannot be undone require pessimistic concurrency control. When neither of these limitations holds, optimism gives a faster, more responsive, more usable application.

## Acknowledgements

Mary Whitton, Diane Sonnenwald, and Challe Hudson helped shape this paper, as did the anonymous reviewers of an earlier version. Funding was provided by NIH NCRR 3-P41-RR02170-15S1: "Interactive Graphics For Molecular Studies and Microscopy supplement to study Tele-Collaboration".

## Appendix: Typical Network Round-Trip Time

All of these measurements were made from Chapel Hill, North Carolina. All used the Internet2 when it was lightly loaded, so are comparable to the times that would be seen under most Quality of Service schemes.

| Destination | Round-Trip Time (ms) |
|---|---|
| Other sites in North Carolina | 10 |
| The Ohio State University | 30 |
| Washington, DC | 40 |
| Microsoft Research, Redmond, WA | 80 |
| KU Leuven, The Netherlands | 110 |

Table 2: Round-trip delay over the Internet2 from Chapel Hill, North Carolina to various sites.

## Appendix: The Logical Clock Algorithm

A logical clock is a construct that allows us to create a total ordering of messages in a distributed system [Lam78, BM93]. Every process maintains a local variable $LC$. Processes are event-driven, with three types of events: send, receive, and internal. Every message sent is given a timestamp, $TS(m)$, equal to $LC(e)$. When an event $e$ occurs, $LC$ is updated:

$$LC = \begin{cases} LC + 1 & \text{if } e \text{ is a send or internal event} \\ max(LC, TS(m)) + 1 & \text{if } e \text{ is a receive event} \end{cases}$$

Events are causally-ordered by their timestamps, which increase monotonically. Once a process has received messages with timestamps greater than

$TS(m)$ from every process, all messages with timestamp $TS(m)$ are "stable" and can be executed.

# References


[BM93] Ozalp Babaoglu and Keith Marzullo. Consistent global states of distributed systems: Fundamental concepts and mechanisms. In Sape Mullender, editor, *Distributed Systems*, pages 55 – 96. ACM Press, second edition, 1993.

[Bro97] Wolfgang Broll. Distributed virtual reality for everyone -a framework for networked vr on the internet. In *Proceedings of the IEEE Virtual Reality Annual International Symposium*, pages 121 – 128, March 1997.

[BWA96] John Barrus, Richard Waters, and David Anderson. Locales and beacons: Efficient support for large multi-user virtual environments. *IEEE Computer Graphics and Applications*, 16(6):50 – 57, November 1996.

[EG89] C A Ellis and S J Gibbs. Concurrency control in groupware systems. In *Proceedings of the ACM SIGMOD '89 Conference on the Management of Data*, pages 399 – 407, 1989.

[Fun95] Tom Funkhouser. Network services for multi-user virtual environments. In *Network Realities*. IEEE, October 1995.

[GUN96] 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*, Seattle, WA, 1996. ACM.

[Her87] Maurice Herlihy. Concurrency versus availability: Atomicity mechanisms for replicated data. *Transactions on Computer Systems*, 5(3):249 – 274, 1987.

[Her90] Maurice Herlihy. Apologizing versus asking permission: Optimistic concurrency control for abstract data types. *Transactions on Database Systems*, 15(1):96 – 124, 1990.

[HSM+00] Thomas Hudson, Diane Sonnenwald, Kelly Maglaughlin, Mary Whitton, and Ron Bergquist. Enabling distributed collaborative science. Video Proceedings of ACM Conference on Computer-Supported Collaborative Work, 2000.

[Hud99] Thomas Hudson. Transformational concurrency control for collaborative design and arrangement. Technical Report TR99-025, UNC Chapel Hill, 1999.

[KP88] Glenn E Krasner and Stephen T Pope. A cookbook for using the model-view-controller user interface paradigm in smalltalk-80. *JOOP*, pages 26 – 49, 1988.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commmunications of the ACM*, 21(7):558 – 565, 1978.

[Lan86] K A Lantz. An experiment in integrated multimedia conferencing. In *Proceedings of Conference on Computer-Supported Cooperative Work*, pages 267 – 275, 1986.

[LJDea99] Jason Leigh, Andrew Johnson, Tom DeFanti, and et al. A review of tele-immersive applications in the cave research network. In *Proceedings of IEEE VR99*, March 1999.

[Mee98] Michael Meehan. Information sharing in collaborative virtual environments. In *2nd Annual Workshop on System Aspects of Sharing a Virtual Reality, Collaborative Virtual Environments*, June 1998.

[Mee99] Michael Meehan. Survey of multi-user distributd virtual environments. In *Course Notes: Developing Shared Virtual Environments*. ACM Press, August 1999.

[MF96] Blair MacIntyre and Steven Feiner. Language-level support for exploratory programming of distributed virtual environments. In *Proceedings of the ACM symposium on User interface software and technology*, pages 83 – 95, November 1996.

[Mil96] D Mills. Simple network time protocol (sntp) version 4 for ipv4, ipv6, and osi. Technical Report RFC 2030, Internet Engineering Taskforce, 1996.

[MZP+94] Michael R Macedonia, Michael J Zyda, David R Pratt, Paul T Barham, and Steven Zeswitz. Npsnet: A network software architecture for large-scale virtual environments. *Presence: Teleoperators and Virtual Environments*, 3(4):265 – 287, 1994.

[MZP+95] Michael R Macedonia, Michael J Zyda, David R Pratt, Donald P Bru tzman, and Paul T Barham. Exploiting reality with multicast groups. *IEEE Computer Graphics and Applications*, pages 38 – 45, 1995.

[Pra99] Atul Prakash. Group editors. In Michel Beaudouin-Lafon, editor, *Computer Supported Co-operative Work*, Trends in Software, pages 103 – 133. John Wiley & Sons, 1999.

[RNRG96] Matthias Ressel, Doris Nitsche-Ruhland, and Rul Gunzenhauser. An integrating, transformation-oriented approach to concurrency control and undo in group editors. In *CSCW*, pages 288 – 297, Cambridge, MA, 1996. ACM.

[RRD90] M A Ranft, S Rehm, and K R Dittrich. pages 575 – 583. IEEE, 1990.

[SBM+01] Diane Sonnenwald, Ron Bergquist, Kelly Maglaughlin, Eileen Kupstas Soo, and Mary Whitton. Designing to support colalborative scientific research across distances: The nanomanipulator environment. In Elizabeth Churchill, Dave Snowdon, and Alan Munro, editors, *Collaborative Virutal Environments*. Springer Verlag, 2001.

[Sch93] Fred B Schneider. Replication management using the state-machine approach. In Sape Mullender, editor, *Distributed Systems*, pages 169 – 197. ACM Press, second edition, 1993.

[SSS99] Christian Schuckmann, Jan Schummer, and Peter Seitz. Modeling collaboration using shared objects. In *Proceedings of the*

*international ACM SIGGROUP conference on Supporting group work*, pages 189 – 198, Phoenix, AZ, 1999. ACM.

[SW94] D Snowdon and A West. Aviary: Design issues for future large-scale virtual environments. *Presence: Teleoperators and Virtual Environments*, 3(4):288 – 308, 1994.

[SZ99] Sandeep Singhal and Michael Zyda. *Networked Virtual Envorinments: design and implementation*. ACM Press, 1999.

[TCO+97] Russell Taylor, Jun Chen, Shoji Okimoto, Noel Llopis-Artime, Vernon Chi, Fred Brooks Jr., Mike Falvo, Scott Paulson, Pichet Thiansathaporn, David Glick, Sean Washburn, and Rich Superfine. Pearls found on the way to the ideal interface for scanned-probe microscopes. In *Proceedings of IEEE Visualization '97*, pages 467–470, October 1997.

[WB94] Colin Ware and Ravin Balakrishnan. Reaching for objects in vr displays: Lag and frame rate. *ACM Transactions on Computer-Human Interaction*, 1(4):331 – 356, 1994.

[Wei93] William E Weihl. Transaction-processing techniques. In Sape Mullender, editor, *Distributed Systems*, pages 329 – 352. ACM Press, second edition, 1993.