Datura: distributing activity in peer to peer collaborative virtual environments

Christopher Derek Just
Iowa State University

Follow this and additional works at: https://lib.dr.iastate.edu/rtd

Part of the Computer Sciences Commons

Recommended Citation
Just, Christopher Derek, "Datura: distributing activity in peer to peer collaborative virtual environments " (2004). Retrospective Theses and Dissertations. 955.
https://lib.dr.iastate.edu/rtd/955

This Dissertation is brought to you for free and open access by the Iowa State University Capstones, Theses and Dissertations at Iowa State University Digital Repository. It has been accepted for inclusion in Retrospective Theses and Dissertations by an authorized administrator of Iowa State University Digital Repository. For more information, please contact digirep@iastate.edu.
Datura: Distributing activity in peer to peer collaborative virtual environments

by

Christopher Derek Just

A thesis submitted to the graduate faculty
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Major: Computer Science

Program of Study Committee:
Carolina Cruz-Neira, Major Professor
Robyn Lutz
James Oliver
Wallapak Tavanapong
Johnny Wong

Iowa State University
Ames, Iowa
2004

Copyright © Christopher Derek Just, 2004. All rights reserved.
This is to certify that the doctoral dissertation of

Christopher Derek Just

has met the dissertation requirements of Iowa State University

Signature was redacted for privacy.

Major Professor
Signature was redacted for privacy.
For the Major Program
Table of contents

List of figures ........................................................................................................ vii
List of tables ........................................................................................................... viii
Acknowledgements ............................................................................................... ix
Abstract .................................................................................................................. x

1. Introduction ........................................................................................................ 1
   1.1. Defining Collaborative Virtual Environments ........................................... 4
       1.1.1. General VR definitions ......................................................................... 4
       1.1.2. Collaboration definitions ..................................................................... 5
       1.1.3. A point of clarification: collaboration versus clustering .................... 5
   1.2. CVE applications ......................................................................................... 6
       1.2.1. Virtual teleconferencing ...................................................................... 7
       1.2.2. Collaborative walkthroughs .................................................................. 9
       1.2.3. Collaborative design reviews and collaborative design ....................... 10
       1.2.4. Training applications and simulators ................................................... 12
       1.2.5. Entertainment applications .................................................................. 13
   1.3. Scope and goals of research ........................................................................ 14

2. Technical requirements and problems in CVEs ........................................... 18
   2.1. Networking ................................................................................................. 18
   2.2. Distributing the state of entities .................................................................. 20
   2.3. Area of interest management ...................................................................... 21
   2.4. Entity control and ownership .................................................................... 22
   2.5. Creating the environment .......................................................................... 23
   2.6. Saving the environment ............................................................................. 24
   2.7. Location of computation ............................................................................ 25
   2.8. User communication and avatars ............................................................... 27
   2.9. Presentation ............................................................................................... 29
   2.10. Summary of technical requirements ......................................................... 30

3. Previous work in CVEs .................................................................................... 31
   3.1. Network-centric tools ................................................................................ 31
       3.1.1. General network libraries: ACE and Plexus ...................................... 31
       3.1.2. Distributed object technologies ......................................................... 32
       3.1.3. NPSNET, HLA, and VRTP ................................................................. 33
   3.2. CVE libraries with simple entity management .......................................... 34
       3.2.1. DSO ................................................................................................. 35
       3.2.2. CAVERNSOFT ................................................................................. 35
       3.2.3. DIVE ............................................................................................... 36
       3.2.4. DOVRE .......................................................................................... 37
   3.3. Advanced entity manipulation .................................................................... 37
7. Controlled transfers of location ................................................. 92
  7.1. Goals of the transfer procedure ........................................ 92
    7.1.1. Preservation of the master state ................................ 92
    7.1.2. Correct functioning of all other mirrors at all sites ..... 94
    7.1.3. Preservation of partial computations .......................... 95
    7.1.4. Minimal disruption for users .................................. 95
  7.2. A strategy for performing role transfers ............................ 96
    7.2.1. Transferring the role ....................................... 96
    7.2.2. Maintaining consistency of other mirrors .................. 98
    7.2.3. Considerations for Processing Group maintenance .......... 100
  7.3. Concrete examples ..................................................... 101
    7.3.1. Initiating a controlled transfer ............................. 101
    7.3.2. Completing a transfer ....................................... 104
  7.4. Performance effects of controlled transfers ...................... 105
    7.4.1. Transfer design and unavailable time ........................ 105
    7.4.2. Effects on user-detectable interaction latency .......... 106
    7.4.3. The big picture: Distribution of computation and responsibility 106
    7.4.4. Location Manager metrics for packaging and transferring state 109
  7.5. Checkpoint .................................................................. 109

8. Uncontrolled transfers of location ......................................... 111
  8.1. Causes of uncontrolled transfers .................................... 111
  8.2. Detecting vanishing sites ............................................. 112
    8.2.1. Location Manager heartbeat .................................... 112
    8.2.2. Noticing unanswered messages ................................ 113
    8.2.3. Detecting networking failures ................................. 113
    8.2.4. Verifying vanished sites ..................................... 114
  8.3. Handling vanished PG members ........................................ 114
    8.3.1. Promoting a new processing group member ................ 114
    8.3.2. Managing incorrect promotions ............................... 116
  8.4. Handling vanished masters ............................................ 116
    8.4.1. Beginning succession negotiations ......................... 117
    8.4.2. Succession priorities ......................................... 117
    8.4.3. Finding the highest-priority mirror ........................ 118
    8.4.4. Handling conflicts ............................................ 119
    8.4.5. Declining promotion to Master ............................... 119
    8.4.6. Managing priorities during controlled transfers .......... 120
    8.4.7. Priorities and backup masters ................................ 120
    8.4.8. Other uses of priority numbers ............................... 121
  8.5. Maintaining consistency and resolving state conflicts .......... 121
    8.5.1. Recreating lost state ....................................... 122
    8.5.2. Reconciling multiple masters ............................... 125
  8.6. Concrete examples ..................................................... 126
    8.6.1. Assignment of priority numbers ............................. 127
    8.6.2. Succession negotiations ..................................... 127
    8.6.3. State reconstruction ......................................... 130
  8.7. Performance impact of uncontrolled transfers .................... 131
    8.7.1. Transfer design and unavailable time ....................... 131
    8.7.2. Effects on latency and load balancing ..................... 134
List of figures

2.1. Avatars in action ................................................................. 28
3.1. Development of avatars ..................................................... 47
4.1. A Datura entity that represents a user input device .............. 56
4.2. An inanimate object that can be grabbed by a user ............. 57
4.3. Object creation ................................................................. 62
5.1. Peer to peer distribution .................................................... 68
5.2. Master/Slave distribution pattern ...................................... 70
5.3. Master/Processing Group/Slave pattern .............................. 71
5.4. Initiating manipulation ..................................................... 79
5.5. Performing manipulation .................................................. 80
7.1. Distribution of master mirrors of elements .......................... 108
8.1. Assignment of succession priorities .................................... 135
10.1. Layout of test network .................................................... 146
10.2. Wide-angle view of the combined Template App / Robot Playground .................................................. 147
10.3. A group of movable objects ............................................. 149
10.4. Three "huey" pathfinding robots ....................................... 150
10.5. Gravbots in their arena ................................................... 151
10.6. Initial locations of objects ................................................ 152
List of tables

2.1. CVE requirements summary ................................................................. 30
3.1. A summary of how previous research has addressed the requirements of CVE software ....... 50
7.1. Examples of master state treatment during transfer ...................................... 94
10.1. Number and sizes of messages created by each site during a 10 second interval .......... 155
10.2. Latency times for initiating interactions .................................................. 156
10.3. Effects of distribution pattern on pathfinder performance ............................ 158
10.4. Update times at each site during pathfinding ........................................... 159
10.5. Distribution of roles among sites before and after load balancing ...................... 160
10.6. Role distribution before and after balancing, when site 4 becomes overloaded .......... 161
10.7. The order that elements are selected for transfers depending on users' location ........ 162
10.8. Effects of removing Site 3 ..................................................................... 164
10.9. Messages sent during succession negotiations when Site 3 is removed ............... 164
10.10. Variation in unavailable times for elements undergoing succession negotiations .... 165
10.11. Distribution of roles before and after a net split ...................................... 166
10.12. Recovery times for pathfinder robots when interrupted while following a path ...... 175
"And when you feel you're near the end..."

But I didn't get there alone. It's been an incredible privilege being involved with VRAC for the past several years, and I'd like to thank everyone I worked with in that time. It wasn't always easy, but it was never dull.

I never would have gotten even this far without the support of my friends. Thanks, especially, to everyone who's picked up a d20 with me in the last few years. When it inevitably came time to sacrifice productivity for the sake of continued sanity, you guys were there to help me blow off steam.

Finally, I'd like to thank my family. My parents have been waiting for this for a long time, even when I was too busy enjoying the journey. This one is for them.
Abstract

Collaborative Virtual Environments (CVEs) are an exciting advance in the field of Virtual Reality (VR) research. By joining VR systems – and users – at widely scattered geographic locations, VR changes from an isolated experience to one of communication, interaction, and collaboration. Much research effort is being placed into the development of tools and techniques to power these collaborative experiences.

This dissertation describes the Datura toolkit for CVE development and, more importantly, the new concepts and methods that make Datura unique. We focus on the idea of Location of Computation (LoC) – methods for determining where, among all the sites participating in a CVE, particular calculations or particular decisions should be made. Datura connects sites into a peer to peer network, allowing each one to participate fully in bringing the virtual world to life.

Datura works at the level of elements – individual components that imbue shared objects with data, behaviors, and capabilities. These elements are shared among all sites, and control over them can be granted or migrated individually. This dissertation discusses the mechanisms for transferring control and computation, and provides a system for deciding where control should reside for each element in a CVE. An extensive set of tests and evaluations are also described, verifying the capabilities of the Datura system and demonstrating the performance and error-handling gains that are made by this fine-grained control over the location of computation.
1. Introduction

Virtual Reality (VR) has evolved from a scientific fantasy into an active, rapidly advancing field of computer science research. The next and perhaps most exciting stage of that evolution is the transformation from isolated, single-user worlds into highly scalable, massively networked Collaborative Virtual Environments (CVEs). These CVEs can unite geographically separated users in a single immersive illusion of place. The users might be engineers working on a single design, artists performing a single play, or even gamers playing in a single game.

There have been numerous efforts to create software to support CVEs. Many of the first efforts were simple networking toolkits, customized for individual applications and their specific interactions. Most of these toolkits provided very low-level APIs for network communications. They left all the details of network use to the applications themselves, creating a burden for the developer and limiting the deployment and versatility of CVEs. More recently, these software libraries have become more aware of the needs of the applications, such as the need to handle communications between users, or the need for users to manipulate elements of the environment. One recent trend in CVE research is the design of libraries of specific "behaviors" that can be applied to objects and users in collaborative spaces.

The work presented in this thesis focuses on methods to enable CVEs in a peer-to-peer (p2p) network of sites. This presents a number of technical challenges. A key problem is that the various sites participating in the CVE may have widely disparate computational resources, and therefore some sites might be at a disadvantage if complex worlds are being shared. This problem has frequently been addressed by performing most computations for the environment on a centralized server machine (as we shall see in Chapter 3), but in a peer-to-peer system, no such server exists.

The lack of a centralized server has other repercussions as well. For example, there is not any well-defined central authority to resolve questions of synchronization (e.g. which of two users grabs a
particular object first). Nor is there a single place that a new site can query in order to join into a CVE and learn about the places, objects, and users it contains. Also, individual sites and users could join or leave the CVE at any time, whereas a server would provide a consistent point that exists throughout the lifespan of a CVE.

To address these issues, our particular emphasis centers around handling the Location of Computations (LoC) for a CVE. Our approach is to investigate methods for performing calculations at the sites best able to handle them, and then distribute the results to all other sites in the collaborative session.

This control over the location of computations provides a number of advantages. For example, it may be used to enhance the performance of the CVE. A user may trigger an action that requires a numerically intensive computation that can't be performed with his local resources, but which could easily be performed on another site with faster processing abilities. A similar case exists for computations that require access to a large amount of data – it may be easier to perform the computation on another site with local access to the data than to send all the data to the user's own computer.

Conversely, there are times when it is advantageous to perform a computation locally. If the user is attempting to physically interact with objects in the virtual environment – performing an assembly task, for example – the latency of the environment's reaction to user input can be an important concern [Ware94]. It may be better to perform computations related to such a task locally rather than suffer the additional lag imposed by the network.

Controlling location of computation can also provide a methodology for handling the synchronization and authorization issues that would otherwise require a centralized server. One site can be designated to handle any computations that involve ownership or control decisions for a particular object. Instead of one server deciding everything, each site could become a sort of temporary mini-server, in charge of decisions for a subset of objects in the CVE. When a site leaves
the CVE session, other sites can take over responsibility for its objects.

In short, our approach is to create software components that might choose to do their processing somewhere close to a particular user, or may offload difficult computations to an underutilized node elsewhere. They may spread out certain computations, or they may use multiple sites for redundancy. In particular, they will attempt to survive when any site is removed from the network, even if that site performed the primary processing for a particular element and disappeared without warning.

The flexibility gained by creating this kind of control over computations allows us to look at all sites in the CVE as a shared pool of resources available to everybody in the CVE. In this way, applications can be less concerned by the differences between local and remote resources, and don't have to limit a particular user's actions in the environment due to local resource limitations. Our goal is to provide a number of benefits to help drive the use and deployment of CVEs. In addition to better resource and network usage through controlling the locations of computations, this work can aid in the development of CVE applications by providing well-defined, flexible elements that can be applied to a variety of application areas, as well as providing patterns and examples for designing domain-specific elements.

The following chapters explore the roles of CVEs in a variety of disciplines, and investigate the particular requirements different application areas place on the underlying CVE technology. We then examine a number of previous works in the field, and note the particular techniques they used and requirements they met. With that necessary background, Chapter 4 introduces the basic structure of CVE software we designed for LoC experiments, and its realization in the Datura software library. The second half (chapters 5 through 9) describes the networking and distribution capabilities of such a system, and explains the location-management decisions and methods it uses. The dissertation closes with a series of tests designed to evaluate the correctness and benefits of the LoC control techniques described herein.
1.1. Defining Collaborative Virtual Environments

The words "collaboration" and "virtual reality" have been much abused and often confused. In this section we provide definitions for these and related phrases as we use them in the context of this document. With definitions in hand, we then look at the hardware and software capabilities needed for CVEs, and explore the particular challenges of collaborative VR systems.

1.1.1. General VR definitions

The terms "Virtual Reality" and "Virtual Environment" have been used many ways by many people in many fields. Marketers use the terms to indicate almost any kind of three-dimensional computer display, including arcade games and desktop applications. Serious researchers use them to indicate computerized systems that provide a sense of "immersion" with body tracking and stereo displays. Earnest (but perhaps less serious) researchers have associated Virtual Reality with the immersion of literature or even of visionary experiences [Rushkoff95].

In this document, "VR" refers to an interactive, immersive, computer-generated environment. An immersive system is one that gives the user (or users) a sense of presence within an artificial world [Burdea/Coiffet94a]. A "VR System" is the hardware and/or software used for presenting VR. Many modern VR systems use head-mounted displays (HMDs) or surround the user with projection screens [Cruz-Neira+93] to provide an all-encompassing, stereo view of the world. They use tracking systems, wands, and/or gloves to monitor the movements of the user, so that the application can respond to the user's gestures or head movements. At the most basic level, in a VR system a user can move in the virtual world simply by walking around in a tracked space, and pick up a virtual object by selecting it with a wand or grabbing it with a DataGlove [VPL87].

A large number of toolkits have been presented to provide the software side of a VR System. These include VR Juggler [Bierbaum+01], CAVElib™ [Cruz-Neira95], DIVERSE [Kelso+02], and MR Toolkit [Shaw93], to name just a few well-known examples.
A "Virtual Environment" (VE) is an instance of a world presented in VR. In this document we will consider a number of different kinds of VEs – different application areas with different particular requirements. A VE could represent anything from an architectural walkthrough of the Taj Mahal to a training simulation of the surface of Mars.

1.1.2. Collaboration definitions

A "Collaborative Virtual Environment" (also, occasionally, "Networked Virtual Environment") is a VE in which multiple users, sometimes widely separated geographically, can interact with each other and with the items in the environment. Zyda and Singhal define a CVE as giving users several primary features: a shared sense of space, presence, and time; a way to communicate with each other; and a way to share and interact with each other [Singhal/Zyda99a].

The individual physical locations that communicate with each other to create a CVE are called "sites". A site is a single VR system; it is most often a single computer with a single user. In the case of VR systems powered by a cluster of computers, typically only one will actually communicate with the other sites in the CVE.

Another ambiguous term when discussing VR, and especially collaborative VR, is "object". This might refer to a visual object such as a chair, displayed by the VR system using 3D graphics. On the other hand, it might refer to an object in the sense of a software object, such as an instance of a C++ or Java class. For clarification, this document uses the term "object" to refer to the former kind: the graphical things that are presented to the user in a VE, and with which the user can interact. "Entity" is used for the sense of a software object: a data structure with associated algorithms.

1.1.3. A point of clarification: collaboration versus clustering

There are currently two main branches of interest in using networks of computers for VR: collaboration, as defined above, and clustering. These branches have a few similarities and some fundamental differences, and so it may prove helpful to explore their (dis)similarities, and see why they need different software approaches.
Clustered VR systems use multiple computers to create a virtual environment in a single physical location [Allard+02][Schaeffer/Goudeseune03][Olson02]. For example, the computers might power the individual displays of a surround-screen system. Alternately, one computer might provide the graphics for an HMD, while a second machine manages the IO devices, and a third handles computations and simulation. Many such combinations are possible.

The difference is that a CVE uses multiple computers to provide displays to different users in different geographic locations, while a cluster uses several machines to create a single display for users at a single location. The advantages of using a VR cluster include scalability (by adding additional computers to the cluster) and the ability to use conventional PC hardware instead of a single high-end machine. Of course, both levels of networking might be used simultaneously - two VEIs each powered by a cluster of PCs might use CVE software to provide communication and interaction between their users.

While there are superficial similarities, the software required to run collaborative and clustered systems have very different requirements. Clustered systems generally have much tighter synchronization constraints and very low tolerance for communication latency and network jitter. For example, clustered software solutions typically lock all machines in the cluster to the same frame rate, and all machines are synchronized to swap frame buffers simultaneously - the individual computers are often genlocked together for this purpose. This level of synchronization is impossible in a CVE spread across a large geographic area, especially if it uses public networks for its connections. The latency of communication is much higher, and the CVE software must try to provide a reasonable extrapolation of its limited, dated knowledge of the state at each site. CVE software must also provide communications between users, for example with audio and visual avatars.

1.2. CVE applications

The first VR systems were single-user systems. Due to hardware limitations of the computers of the time, early systems using head-mounted displays could only support a single HMD. Later,
surround-screen systems allowed multiple users in the same physical place to view the experience simultaneously. However, usually only one person had control or interaction with the environment¹.

People are naturally gregarious – we tend to work in teams, and to seek help and feedback from our peers. Because of this, researchers have long sought to enable this kind of teamwork by uniting people in remote locations in a single virtual environment. Meanwhile, science fiction authors have envisioned a future where millions could participate in a single shared reality [Gibson84] [Moorcock94], subsuming most other forms of long-distance communication.

While the science fiction view remains a long way off, research has brought CVEs into a number of practical application areas. The next several sections explore some of these uses of CVE technology, and show the potential value in bringing together groups of people separated by geography. The applications are arranged in approximate increasing order of interactivity and technical complexity. The different sorts of interactivity required for different uses bears particular consideration. In CVE-based teleconferencing applications, interactivity is primarily person-to-person with the VE acting as a medium. The walkthrough and design review applications described later feature more interaction between people and the virtual environment itself. Continuing into application areas such as collaborative design and training, interactions between entities in the virtual environment, and between users and those entities, will become more apparent.

1.2.1. Virtual teleconferencing

One of the first, and comparatively simple, uses of CVEs, was in virtual teleconferencing applications. Most readers will be familiar with non-VR forms of teleconferencing. Videoconferencing, for example, combines video and audio from all the connected sites. The video provides a more personalized experience, with a greater sense of presence and involvement than a

¹ There have been a few exceptions. In recent years, it has become technically feasible to power multiple HMDs with a single powerful computer, and several methods exist for providing multiple tracked views in projection-based VR systems [Blom02] [Agrawala+97].
conference phone call, and allows the participants to use some body language.

On desktop PCs, software such as Microsoft's NetMeeting [Microsoft03] and other applications using the T.120 standard [ITU96] have been used to implement nonimmersive teleconferencing. In addition to integrating video and audio communications into a computer display, NetMeeting provides several capabilities for sharing applications and the computer interface. For example, it includes the idea of a shared "virtual whiteboard" - a window that users can write or sketch in, with the contents shared with all the other users [Fraunhofer99]. These applications can also share traditional GUI applications. They do this by transmitting screen captures of an application's window. Remote users can even take control of an application - NetMeeting can capture mouse and keyboard input and inject it into the input handler for the application.

Another step in the direction of VR teleconferencing is presented by various web-based 3D "chat" technologies, such as Adobe Atmosphere [Adobe], WorldsPlayer [WorldsPlayer], or ActiveWorlds [ActiveWorlds]. These projects drew their inspiration from technologies like Internet Relay Chat (IRC) [Oikarinen] and the chat rooms that can be found on computer BBS systems. Although these applications focused on text instead of voice communication, they added visual representations of users (called "avatars") and provided virtual locations in which the avatars could mingle. In many of these efforts, the environment existed primarily as scenery, and interaction was limited to gathering avatars together in groups to talk.

Teleconferencing in CVEs combines the ease of voice communication with the advantages of giving participants virtual avatars. In such an immersive teleconferencing application, all participants are given the illusion of being present in the same location, which might be a simple model of a conference room or something more fanciful [Ståhl/Andersson] [Ståhl99]. The avatars and voice communications allow a wide range of interactions to occur. Participants may be able to gesture dramatically to accentuate their points, or stand in a position of authority at the head of a virtual conference table. Users may be able to break up into smaller groups for private discussions. Unlike a
real gathering, the CVE might allow users to control the communications channels to prevent eavesdropping.

There has been much research in various kinds of avatars with various levels of expressive power; section 2.8 gives a summary of current approaches and research efforts in this direction.

Many CVEs designed for teleconferencing have adapted whiteboards and other visual communication tools from the nonimmersive applications described above [Stähl92]. Frequently, the emphasis is on utility and not on a seamless illusion of presence. For example, a participant does not have to pick up a virtual dry-erase marker to use the virtual whiteboard - he might simply assume control and then add a message using a keyboard and mouse, or a voice recognition system, or a handheld computer [Hartling+02].

Teleconferencing CVEs have also been demonstrated at scales that would make traditional desktop videoconferencing awkward. [Greenhalgh+99] provides an interesting example, in which a CVE was used to conduct a sort of virtual quiz show with a large number of people entering and leaving the environment. It also demonstrated the use of television broadcasts of images from the CVE in order to involve an unlimited number of passive observers.

Perhaps more so than any other kind of CVE application, virtual teleconferencing solutions are built around the idea of participants seated at desktop computers with nonimmersive displays. While this effects the realism of the environment, it also greatly increases the number and kind of users who can participate.

1.2.2. Collaborative walkthroughs

In the telecommunications-centered applications described above, the priority is on communication between the participants; the environment - the background, the furnishings, etc. - is usually treated as an afterthought. Another use of CVEs is to provide a collaborative walkthrough, for example of an architectural model [HITL]. This can provide a way for architects and designers to provide an immersive, realistic vision of a project while still in its early design stages [Schmitt+95].
Alternately, the environment could be used to give a guided tour through an environment such as a museum [Beckhaus+01], or as a historical presentation of a location that no longer exists [Moltenbrey01] [Wright00]. In each case, the networking aspect of collaboration technology may be used to present the model to clients in multiple remote locations.

In a walkthrough of this sort, the environment and its realistic presentation becomes the primary focus of the application. The ability to provide an immersive experience to each of the users becomes even more important than with teleconferencing applications. The immersive experience can make a model much more compelling. It can also have a significant effect on the participant's ability to perceive the structure and layout of the model².

All the communications abilities in teleconferencing applications can apply to these sorts of walkthroughs as well, including the need to have private conversations between small groups, or the ability to take notes on a shared whiteboard. It may also be important to have additional ways to access information about the building or environment (historical details, construction or materials notes, etc.) [Guidazolli00]. Avatars can have additional uses in these environments. A humanoid figure can provide a sense of scale in a room; the presence of a group can show how well people fit in a particular space. Avatars may be able to point at features of the environment, or lead the other participants on a tour. It may also be useful to allow one user to take control (or partial control) of the other users' avatars, effortlessly carrying them along on a tour while still allowing them to look around at whatever draws their attention.

1.2.3. Collaborative design reviews and collaborative design

Of course, architectural models are not the only kinds of designs that might be reviewed in a CVE. The same application framework might be extended and enhanced for looking at CAD data for

² Some experimental results [Henry92] suggest that virtual environments are well-suited to understanding qualitative features, such as the layout of an architectural model, and less effective at communicating quantitative aspects such as size and distances (at least when HMD-based VR systems were
a new automobile or aircraft, or examining a single small component thereof, or even for looking at the molecular structure of a new chemical or drug. Collaborative design reviews have been a feature of several commercial applications, including Vis Concept [EDS] and DIVISION Reality [PTC].

A situation like this shares many of the requirements of a walkthrough as described above. The same needs for communication and sharing information exist. The ability to view the model or data in an immersive environment with free movement is still important, especially if the objects under consideration are life-size representations - an automobile body, for instance.

There are additional requirements. Engineers using a virtual environment for design reviews would like to be able to manipulate and change the models as they are working with them. For example, they might want to take apart the components of an engine to look inside, or to better see how they fit together - or even if they can fit together in the real world. They might want to display a cutaway view of one or more pieces of a model, or render certain parts temporarily invisible or transparent. They might want to attach annotations to certain pieces of the model.

The engineers in question might even want to make the step from "design review" to design itself - directly modifying the models and underlying data. Users of an immersive design application could change the color of one part of a model or the thickness of another, or subtly alter the curve of a car's roof. They might wish to rearrange the buttons on a control panel, or lower the rear view mirror to provide better visibility. This level of modeling in immersive environments is still a topic of ongoing research, as seen in projects such as DivEdit [Stenius96] and NIME [Yoshimori00].

In short, design review adds the need to manipulate data to the viewing and communications requirements we've already discussed. In a CVE, we need to share those manipulations, and we need to keep users from interfering with each other. A CVE application for design also requires some method for persistence: a way to save whatever changes were made inside of it.

Recent experiments [Arms02] suggest that this is very dependent on the particulars of display devices.
1.2.4. Training applications and simulators

Many VR applications, collaborative or not, have been designed as training applications. There are a number of factors that make VR training particularly appealing. In some cases, traditional training is prohibitively costly - flight simulators [Menendez99][CG2] are one example of how VR can address that problem. Other times, training with a real environment or real equipment can be hazardous for the trainee (for example, when working with military equipment [McDonough92] or weapons systems [Burdea/Coiffet94b]) or others (as in a virtual surgery practice session [Kühnapfel97]). NASA uses VR to simulate space environments and train for procedures on shuttle missions [Nasa93].

Collaboration has a number of uses in such a virtual environment. For example, it can allow a teacher and one or more students to interact in the same space, even if they are physically remote in reality. It can also be used to link different groups of students - for example, joining several combat flight simulators for a virtual dogfight. [Slater00] describes a system allowing actors to rehearse collaboratively before meeting for a performance.

Training applications may need several of the capabilities described above - depending on the particular task, teacher and students may need to navigate through virtual terrain or architecture, or they may need to interact with highly-detailed models of machine or vehicle components. At the same time, it can become more important that objects in the environment act in realistic ways. For example, objects may need to fall to the floor when dropped, or they may need to be held by particular points at particular angles in order to be used effectively. Research has indicated a relation between the user's sense of presence in the environment and the ability to perform complex tasks such as training exercises. [Mania00].

The teacher in a training application may need special capabilities. For example, he may...
need the "guided tour" capacity described above, or he may need to take over an object that a student is manipulating. He may need to reset the application to its initial state, or trigger some set of events in the environment (such as a canned demonstration of the actions a trainee is supposed to take).

1.2.5. Entertainment applications

VR has stepped, slowly and uncertainly, from the lab into entertainment, ranging from the now-crude Dactyl Nightmare [Virtuality91] to state of the art installations at DisneyQuest [Shortal99]. Even home users are starting to experience some aspects of immersive VR, as force feedback devices become popular, and some consumer video card suppliers are including active-stereo glasses. For home users, networking has become a vital part of the computer gaming landscape. Almost every game includes a "multiplayer" mode, which might allow players to cooperate to achieve the game's goal - or more likely will just allow the players to shoot at each other. Attractions such as DisneyQuest's Aladdin magic carpet ride [Pausch+96] allow small groups of users to compete head-to-head in a single distributed immersive environment.

While the true intersection of these multiplayer (and massively-multiplayer) games with fully immersive VR (i.e. with head and body tracking and full-surround displays) lies somewhere in the future, recent years have seen increased communication between the entertainment and VR research communities [Hecker00]. Many of the issues in networked games might apply in the future to any large-scale virtual environment; they have been brought to the fore in gaming circles because of the size and dedication of the user base and the need to supply a maintainable, stable, end-user-ready software system.

There are many kinds of networked games; those that border on VR territory include first-person shooter (FPS) games like Quake [id96] and Unreal Tournament [Epic02], and networked or online role-playing games including Neverwinter Nights [Bioware02], Ultima Online [Origin97], and EverQuest [Sony00]. These games share requirements with the training applications and walkthroughs described above. For example, the world must feel real to the user. This may involve
the quest for photorealistic accuracy, or for more abstract verisimilitude - in entertainment the goal is usually the "willful suspension of disbelief." In any case, many games involve large, realistic (if fantastic) worlds or environments, typically with a great deal of support for customization.

Many games are quite demanding on their networks - when two users are shooting at each other, it makes a great deal of difference which laser blast hits first, and which site gets to make that decision.

Games have various levels of interactions between users. FPS games have gradually incorporated greater degrees of teamwork between groups of players, and massively-multiplayer role-playing games concentrate heavily on player interaction. These interactions might include conversing with other users, fighting them, trading with them, or even trying to pick their pockets. Players may form into informal groups or organized teams. They may need the ability to have private conversation between team members, but it might also be part of the game for opposing players to eavesdrop when circumstances allow.

Similarly, the environment of the game may be modifiable in various ways. This might be limited to picking up weapons or ammunition, or it might include making structural changes to the world (e.g. blowing up a wall with the aforementioned weapons). It might even include interacting with the world in a realistic way, such as digging with a shovel or baking bread in a virtual oven.

1.3. Scope and goals of research

In this chapter, we have defined the nature of immersive VR and of Collaborative Virtual Environments. We have explored a variety of application areas in which CVE technology may be used, and considered some of the interactions that may take place between users and the virtual environments. The technological requirements for these interactions, and the approaches taken to them in current CVE research, are discussed in the next several chapters.

After describing the current state of affairs in CVE research, we introduce this dissertation's novel work, which considers the role of location for the many computations that are part of a CVE
software system. The design revealed in the middle chapters of this document presents a new, generalized technique for managing these computations in a peer-to-peer network of sites united to create a single virtual environment.

The research in this thesis is composed of several distinct parts:

- An evaluation of the literature, to ascertain the current capabilities, theories, and methodologies used in the creation of collaborative virtual environment software. The most important developments in CVE research are described in Chapter 3.

- An analysis of the technological requirements of a CVE toolkit, both in general and for specific applications. Chapter 2 provides a discussion of the issues and needed capabilities for CVE software. At the end of Chapter 3, we return to this list of capabilities, and summarize which previous CVE efforts have particularly addressed which requirements, and point out areas that are especially open to further research.

- A method for creating CVEs in dynamically changing peer-to-peer networks of hosts. The design is based on various elements of prior CVE research – for example, the organizational approach of defining objects in the CVE as entities whose behavior is defined by a composition of generalized elements – and draws on several extant projects in peer-to-peer networking. The overall design of this system is outlined in chapters 4 and 5.

- Methods for controlling location of computation for an entity's behaviors, with three distinct goals. First, to provide a way to sustain the CVE as new sites join a session and other sites leave. Second, to provide an effective use of computing resources by performing computations for behaviors at sites with available resources (e.g. offloading computations to sites with faster processors, or balancing loads across sites). Third, to localize or distribute computations in
order to improve the user's experience of the CVE (e.g. reducing latencies in interactions, or providing smooth visual presentations without constant network updates). Chapters 6 through 9 describe the efforts in this area.

- Finally, an evaluation of the effectiveness of these Location of Computation techniques, demonstrating how, and to what extent, the three goals are met by the LoC techniques as provided in a sample implementation. These evaluations are described in the final chapters.

In addition to the research described above, there are a number of additional areas that could be explored in the course of developing capabilities for p2p CVEs. Some of these rely heavily on outside areas of expertise (security, multimedia, etc.). Each of them could warrant full research projects in their own right, and could be valuable follow-up projects to the work presented herein. These topics include:

- **Voice and video communications.** In a fully-functional CVE, there is almost always a need for users to communicate via speech, and we will mention several research projects integrating live video into VEs. This ability could exist alongside the sort of collaborative LoC framework described by this research, with possible intersections in the area of avatar presentation and management. For now, those intersections will be left to future development.

- **Area of Interest management.** Managing Areas of Interest (described fully in Chapter 2) is a major topic in CVE research, with a number of well-established techniques. Exploring how a p2p environment affects these techniques may prove an interesting topic for further research.

- **Security by encryption.** In Chapter 2, we will discover the role of encryption as a defense against passive observers, or against attackers who attempt to inject data into a CVE's communications channel. However, encryption itself is
properly addressed at the network software level. While an important aspect of implementation, encryption of communication channels does not play a role in the design of LoC methods *per se*. Other aspects of security, however, are specifically in the bounds of this LoC design. These include the responsibility for determining whether a particular user can manipulate a given entity, or deciding which set of sites could be trusted to perform a given computation.
2. Technical requirements and problems in CVEs

Chapter 1 explored the high-level uses for collaborative virtual environments; next we look in depth at the specific technological challenges required to provide those interactive experiences to end-users. Following this discussion, we will explore some previous projects in CVE technology, and see concrete examples of how these requirements were addressed.

2.1. Networking

Networking technology lies at the bottom of every CVE implementation. Even at this basic level there are a number of challenges that must be addressed to create a robust and stable collaborative system.

The first problem is establishing the connections between the sites of the CVE, and determining the overall network structure of the application. Most of the CVE projects that Chapter 3 explores use a client-server architecture. Others use a centralized server for synchronization and decision making, but augment this with multicast communications between all sites for position updates or audio/video streaming. Despite recent interest in peer-to-peer (p2p) tools for other fields, there has been relatively little work applying p2p principles to CVE software.

The choice of structure affects how sites find each other: in a client-server environment, each client will simply try to connect to the central server. A p2p network provides other options. A site joining the CVE session can connect to any known pre-existing site [Limewire02a]. A technique used in some p2p applications [Limewire02b] is to look for lists of available sites, and then connect to one or more of them (possibly taking into account load balancing and underlying network performance issues, or connection limitations imposed by firewalls). A common technique in p2p apps is to connect to a handful of other sites, to prevent disconnections as individual sites leave the session.

Closely related to this is the question of the underlying network protocols, particularly the
choice of unicast or multicast messaging. In a multicast setup, a new site only needs to know of a multicast channel to attach to, while in a unicast environment it will need a priori knowledge of (at least) the address of one other site. Of course there are other issues in this choice as well. Multicast can often reduce networking traffic over a unicast solution, but it has limited availability and can also make security more complicated.

The CVE system must also handle the dynamic nature of sites. New sites may enter the environment at any point. They might also leave at arbitrary times, either intentionally or due to some fault in the site or the intervening network. The CVE system must have a policy to determine how new sites introduce entities or users into the environment, and must decide what happens to those entities when a site drops out of the network.

It is at the networking layer that many issues of cross-platform interoperability come into play. Some CVEs have chosen to transmit information in a very fast but machine-specific manner, tying them closely to a single platform. Others have adapted to the requirements of interoperability by dealing directly with issues like structure padding, sizes of data types, and endianness conversions – several toolkits providing these services are discussed in Chapter 3.

Another concern that frequently involves the low-level networking of a CVE is security. Many potential users of CVEs attach great value to the security of their proprietary data, ranging from engineering CAD files to geological survey data to the molecular structures of experimental medicines. Others users just want to avoid cheating at online games. There are two fundamental concerns here: controlling access to data and controlling the ability to modify the virtual environment. At the very least these concerns require encrypted transmission of sensitive data, though the latter (at least) may require authorization and security monitoring at a higher level in the system. This ties in with the choices mentioned earlier: encrypting information will impact overall performance, which might inspire CVE designers to mix encrypted and unencrypted communications to provide security and performance. Managing encryption in multicast communications channels creates problems of its
2.2. Distributing the state of entities

We typically discuss CVEs in terms of the entities that make up the environment. An entity is the software representation of an object in the environment – the floor of a virtual room, or a piece of an engine, or the visual depiction of a user. Most entities have visual representations, but this is not absolutely necessary - it might simply be a software construct that reacts to the state of the environment (e.g. a logging component that records the actions taken by the users around it).

While some entities are static – the representation of a chair, for example - others will actively change their state (a simulation of a running engine, or a virtual robot). And some static entities can change their state with user interaction (a person pushes a chair across the virtual room).

In a CVE, when any entity's state changes, those changes must be transmitted to other participating sites. This has been approached in a number of ways, with tradeoffs between implementation complexity, bandwidth use, and consistency of the environment. A trivial method for managing entity updates is to periodically package the entity's entire state and send it to every other site [Singhal/Zyda99b]. A more bandwidth-efficient technique is to store the entities' state in a central repository and cache that state at other sites, with updates sent out only when a particular element of an entity's state has changed [Anupam94]. Sometimes it is possible to reduce the number of updates sent out by using prediction techniques such as dead reckoning [Capin+97]. There are also techniques to limit the set of sites that need to be informed of any such state change; these are described in the next section.

There are also many possibilities for how to represent data, and how to react to updates. Consider a relatively simple attribute like the location of an entity, and how the CVE can represent its movement. One simple method is to send out updated position information and have every site instantly snap that entity into the new location. Another option is for each site to gradually move its representation of the object into the right location over several frames. While this provides a
smoother experience to the end user, it does so with a sacrifice of correctness in the remote sites' portrayal of the virtual world (since it will take longer for the object to appear in the new correct location).

For many applications, it may be desirable to allow even smarter behavior from their entities. Imagine a virtual toy robot that executes a path-planning algorithm. It might transmit its path to all the other sites, along with an estimated completion time. Then each site can have its representation of the robot follow the path at whatever speed is necessary to complete the path at the proper time. The robot can then proceed on each site without transmitting further messages (at least until something happens to make it reevaluate its path).

In general, the ability for entities to share high-level information about their state changes can provide many opportunities for improving the appearance of the VE while reducing network traffic, and plays an important role in the new research described in this dissertation.

2.3. Area of interest management

Entities may be perceptible to users on several or all sites in the environment. Determining which set of entities a site should receive updates about is an aspect of area of interest (AoI) management. Controlling AoI is an important part of building large-scale CVEs, where maintaining the state for every entity in the environment at every site can require untenable amounts of bandwidth. A more efficient approach is to only keep track of those entities that are relevant for the local users.

AoI algorithms have been the subject of a great deal of research. Many approaches have been based off of a spatial model [Benford93]. These models determine the set of relevant entities by considering an aura in space around the user. The entities included are those in or near the user's field of view, or those that may be able to directly influence the user. Research has also been done to modify the radius of a user's aura dynamically to control the load on the system [Oliveira/Georganas02]. The descriptions of particular CVE projects in Chapter 3 include details of several AoI approaches – see in particular the discussion of Massive.
AoI can also be manipulated to provide specific effects – for example, to create a private communications channel between two users. In this aspect, it has been used to control network multicast channels used for streaming data such as voice.

2.4. Entity control and ownership

In order for users to interact with objects in a CVE, an application has to make decisions about ownership and control of the underlying entities. For example, it may need to determine whether a particular user has permission to push a chair. If two users attempt to grab the chair at the same time, it needs to decide what will happen.

A common method to address these decisions is to use a client-server model for the CVE network (this technique is used by most of the CVE toolkits examined in Chapter 3). When a user on a client tries to manipulate the entity, the server, acting as a central data repository, decides if the manipulation actually occurs [Singhal/Zyda99c]. If many users try to manipulate the same entity at the same time, the server is responsible for determining who succeeds (usually, this is "who got there first"). For a particular entity, the server might be responsible for checking the authority of a user to manipulate it. It might also choose to allow multiple simultaneous manipulations - for example, two workers, each grabbing one end of a virtual steel girder. In any case, the copy of the entity on the server site is the "master" copy, the one with definitive information about the entity's state. Copies of the entity on other sites are often considered as "slaves" or "proxies" - they merely replicate the master's state.

The client-server paradigm is often applied to solve entity ownership and control issues even when the underlying network model is p2p. In this case, the server might simply be an arbitrary node given the task of arbitrating access requests for that particular entity (see the description of Octopus in section 3.5). This idea is heavily expanded upon in the new work introduced in this dissertation, particularly with the notion of dynamically distributing control over individual entities across the network.
2.5. Creating the environment

The previous sections have discussed requirements for managing and controlling the entities in a CVE. These entities must be introduced into the virtual environment in some way. There are two distinct aspects to this problem:

First, some site participating in the CVE must introduce a new entity into the environment. For example, it must tell the other sites that "there is a chair at coordinates (x, y, z)". In a typical client-server system, the server sends this information to each client that connects to it. More sophisticated systems use a technique similar to naming services to allow a top-level server to direct clients to a number of virtual environments, which may be served by the same server or by other sites on the network [Oliveira+Ol]. Even in a client-server arrangement, some entities will be introduced by other sites: the users' avatars are one example.

Second, the data that makes up the entity, such as the 3D geometry file used to draw it, must be transmitted to the other sites. Early networked 3D applications required this data to be downloaded and installed before connecting to a server, but the convenience of downloading these data files directly when joining a session was quickly realized. A technologically more difficult situation is presented when the data to be downloaded is an executable code extension. CVE projects exploring this problem (such as NSPNET [NPSNET] or JADE [Oliveira+00]) most often take advantage of software component models such as Bamboo [Watsen/Zyda98] or Java's dynamic class loading capabilities [Liang/Bracha98].

There are security issues involved with introducing entities into the environment, such as determining which sites have permission to do so. Additional restrictions could include the size of an object or the locations it is allowed to occupy.

---

3 This is demonstrated by trends in networked games, where user-interface issues are major selling points. Games such as the original Quake [id96] and Warcraft [Blizzard94] required data such as character models and maps to be installed by users; their respective sequels [id99][Blizzard02] added automatic
In p2p CVE technology, there are certain additional problems and possibilities that have noteen fully addressed in the research to date. One problem is controlling the introduction of objects into the environment without a central organizing authority. For example, if three sites introduce virtual rooms with different contents, the system must somehow decide how those rooms are connected in the space of the VE. PlaceWorld [Cook01] presents one experimental approach to dynamically laying out areas of interest in a CVE based on user interest.

By contrast, an interesting possibility in p2p CVEs is to take advantage of the "swarm downloading" approach that has become popular in p2p file sharing applications in recent years [Cohen03]. This might reduce the bandwidth load for sites trying to provide new data files or code to other sites in the session, by allowing new sites to download pieces of the data from multiple established sites at once.

2.6. Saving the environment

In many CVE applications, it is important to be able to save the state of the virtual environment, or of some entities in the environment, when a collaborative session ends. For example, users of a collaborative design application need to save their finished designs in the same way users of word processing programs need to save their documents. In CVE research, storing the state of a virtual environment at the end of a session is referred to as persistence.

The first difficulty in implementing persistence is determining a format for storing the data. If the only data that needs persistence is geometric - the results of a design session, for example - there are a multitude of well-established file formats. Most commercial modeling (3ds Max [Discreet02], Maya [Alias03]) and CAD packages (AutoCAD [Autodesk03]) have their own geometry formats. Capturing the full state of an application, including the interactions and behaviors associated with the various geometric entities, is a topic of ongoing research. Development in this
area has led to formats such as VRML [Broll96] and XML-based application description languages such as X3D [X3D] and CONTIGRA [DachseltOl]. There is also research in storing the entire history of activity in a CVE for later playback [Greenhalgh+02].

The second difficulty is determining where to store the persistent data. Again, a client-server architecture gives a ready solution: since the server maintains definitive information about all entities, it can back this up with persistent storage in the form of a database or file system [Singhal-Zyda99c]. In this sense, it acts much like any other kind of file server. This technique can also be used in CVEs that use p2p communication, but it creates a dependence on that single server site always being available. It might be useful to allow persistent storage of a given entity on any site participating in the CVE, but this can lead to a variety of interesting data-versioning issues that have not been addressed to date. For example:

Suppose sites A, B, and C participate in a collaborative design session. The user at site A introduces a 3D model of a car engine, and the users on other sites proceed to make a number of changes. A leaves the session, and the engine model is saved on his local computer. The users at B and C continue, make a few more changes, and then save the model on their own systems. There are now two different versions of the persistent data for the engine. If A later collaborates with another user at a site D, the problem is exacerbated. This question of maintaining consistent data for an entity without having some central storage is unresolved, but parallels might be drawn to the field of version control systems. Most of these use a central server (CVS [Cederqvist02] is the classic example), but recent projects such as BitKeeper [BitKeeper02] endeavor to manage multiple versions of files stored in several servers, without a central authority.

2.7. Location of computation

Some VE applications require complex calculations. For example, they may be running a high-precision physics computation. Entities in the environment might be controlled by path-planning or other artificial intelligence algorithms. If the application is collaborative, decisions must
be made about which sites participating in the CVE will perform these calculations. These decisions can affect not only the total efficiency of the system, but also the workload at individual sites and the quality of interaction for individual users, and even the robustness of the CVE application as a whole. The research challenge is to find good methods for deciding the location for each computation, balancing the capabilities of various sites with the bandwidth limitations of the network to provide the best user experience.

In a client-server system, typically the server(s) will do all the computations. In a p2p network, the simplest solution is for the site that introduces an entity to take responsibility for its computations. This is not always feasible: the site might not have the computational resources or available data to handle the computations. If this is so, responsibility for the computation may have to be offloaded to another site with more processing power or more data storage.

Another solution may be to divide the computation into smaller tasks, and have several sites each contribute to the overall processing task. In this way, entities in CVEs could take advantage of more common aspects of distributed computing to optimize their performance. Even if the bulk of calculations for an entity are only being done at one site, there may be advantages to having more control over which site that is. The simplest example of this is load-balancing, where each site is given approximately an equal amount of work to do over any time period, or an amount proportional to its own capabilities [Murthy/Manimaran01].

Sometimes raw computational speed isn't the most important issue – interaction tasks require the application to respond to the user with as little latency as possible [Ware94]. For example, if a single user is interacting directly with an entity (e.g. pushing a chair around), it may make sense to migrate that entity's workload (collision detection and physics simulation) to the site hosting that user.

When we discussed the need to distribute entity state, we suggested that some of the copies of an entity might be able to perform simple calculations themselves, such as smoothing an animation or following a precomputed path. These calculations have no effect on the master or other copies of the
entity; they only affected the local display of an object.

The ability to migrate the computations required by an entity could also allow entities to persist in the environment, apparently uninterrupted, after the site that previously hosted them leaves the system. If some amount of the entity's internal state is backed up on other sites, it might be able to continue even if the hosting site crashes or is abruptly cut off from the network.

In evaluating previous work in CVEs, we will see that most do not provide this sort of control over the location of an entity's computation. A few systems do, but none explore all the possibilities we have considered. Pursuing these capabilities in depth is the core concept of the research in this thesis.

2.8. User communication and avatars

CVE applications are by definition multi-user. Therefore, it is vital that each site provides some method for representing remote users. Visual representations of human users in VEs are called avatars.

There has been a vast amount of research in creating lifelike avatars. Early CVE research represented users as simple geometric shapes with limited movement or expressive ability. In successive years, avatars have increased in both detail and expressiveness. The most common approach is to represent the user with a highly-detailed 3D model – which may or may not resemble the actual user. Because of the importance of communicating the user's physical actions [Fraser/Glover98], research has focused on ways to make such models highly interactive and expressive [Barrientos/Canny01] and finding ways to deal with a small number of periodically sampled tracked points on the user's body [Capin+97].

---

4 "Avatar" in VR derives from the Hindu term *avataraham* – literally, "the crossing down". An avatar was a physical manifestation of a spirit in the physical plane, and by extension the manifestation of a physical entity in a virtual environment. In computer circles the term was popularized by Richard Garriot's Ultima games [Addams90], in which both the modern and traditional definitions were apropos.
Avatar expressiveness is an important issue: it affects the user's ability to communicate visual details to remote users. [Colburn00] discussed one project detailing the use of eye gaze in avatar communication. In training applications, it can be important to see how a trainer moves when he demonstrates a procedure, or to see which part of a machine he is pointing at. In a game, it can be important to see which direction an enemy is (or isn't) looking.

Another major thread in avatar research concerns the use of live video to represent users. For example, some projects have worked to capture live stereo video of a user's entire body and insert that video into the VE's graphics [Ogi+02]. Other efforts project live video of a user's face onto a physical dummy head [EGM97] or use it as a texture on the head of a computer-generated 3D avatar model (see [Rajan+02] for an example).

Figure 2.1. Avatars in action. A user in a virtual environment (ISU's C6) interacts with a remote user's avatar. Photo copyright ©2000 ISU photo service.
Transmitting live video of users is bandwidth-intensive. One alternate approach is to create a detailed model of a user and then recreate facial movements dynamically based on phoneme recognition of the user's speech [Kshirsagar01].

In addition to sharing visual avatars, users in a CVE will expect to be able to talk to each other. This might involve separate teleconferencing equipment, or transmission of digitized speech along with the rest of the application data. This raises the challenge of deciding which users can hear each other, or which users are allowed to hear each other. This may be solved using user proximity as part of AoI management, or by letting users explicitly control who they are speaking or listening to.

2.9. Presentation

When all the sites are connected, all the entities created and sharing their states as designed, and all the computations are being performed as needed, one major task remains: presenting the CVE to the users.

There are a number of ways to do this. Many CVEs simply present the virtual world as a desktop application with a graphics display window (the early versions of Massive, detailed below, are a good example). Others use web browser interfaces for the same purpose [Broll96].

Of particular interest to the current research are those CVEs that have supported fully immersive VR systems. Some of these have built their own support for visualization, but it has become more common for the CVE software to coexist alongside a separate software system that provides immersive VR services. These services include management of input and output devices as well as visual and audio displays, and may also include a high-performance scene graph for rendering the virtual world. Examples of such standalone VR systems include VR Juggler [Bierbaum+01], DIVERSE [Kelso+02], and CAVELib™ [Cruz-Neira95]. The choice of VR system can affect the overall performance and scalability of the environment - the complexity of the graphical display that can be achieved, for example - and can have major ramifications on the portability and hardware requirements of the CVE system.
2.10. Summary of technical requirements

This chapter examined a number of requirements for CVE tools, based on the various kinds of applications previously listed. Table 2.1 summarizes these requirements, highlighting the ones that particularly apply to the new research described herein. Chapter 3 looks at specific CVE toolkits and the contributions they have made — particularly solutions or partial solutions that have been presented for meeting these requirements.

| Networking          | Find and connect sites.  
|                    | Pass data between sites (client-server or p2p).  
| Distributing entity state | Package object updates for networking.  
|                    | Determining when to send state updates.  
|                    | Determining how to interpolate between updates, or actions while waiting for updates.  
| Area of Interest management | Create strategies for determining which sites need updates about which entities.  
|                    | Managing communications based on AoI results.  
| Entity control and ownership | Check authority and handle synchronization when users attempt to manipulate entities.  
|                    | Determine how control decisions are made (e.g. which sites are responsible for which entities).  
| Creating environment | Introducing new entities into VE.  
|                    | Informing sites about newly-created entities.  
| Persistence | Creating a format for saving entity/environment state.  
|                    | Managing versions in a p2p environment.  
| Location of Computation | Determining where to best perform entity’s computations.  
|                    | Migrating computation/control between sites.  
| User communication | Manage visual avatars for users.  
|                    | Distribute audio and/or video of users.  
| Presentation | Presenting visual representation to user in immersive VR.  
|                    | Receiving user input from VR system.  

Table 2.1. CVE requirements summary. Requirements of particular relevance to p2p environments and the research described herein are italicized.
3. Previous work in CVEs

There have been many CVE research projects in recent years, taking a variety of approaches in implementation and design. Some have been purely research projects, while a few have been put forth as viable end-user (and even commercial) products. Many have been created and abandoned, while some have been the focus of long-term efforts.

In this section, we examine some of these CVE software projects, and consider their methods and contributions to the advancement of CVE technology. The survey concludes with a discussion of in-house work (VRAC's Octopus project) which influenced the current research.

3.1. Network-centric Tools

Many early CVE projects simply concentrated on issues of network setup and connectivity, cross-platform message handling, and so forth. Later projects used separate toolkits for the basic network functionality (ACE, Plexus, CORBA, etc.), or specialized the network communication protocols for the purposes of CVEs (NPSNET et al.). The projects most relevant to our research are described here.

3.1.1. General network libraries: ACE and Plexus

ACE [ACE], the Adaptive Communications Environment, is a software library developed by Washington University. It has a long history of active development. ACE serves as a low-level library for writing cross-platform networked applications. It supports many of the basic technical requirements for network communication between the sites of a CVE. It can mask many of the difficulties of cross-platform development and communication, and provides abstractions for a variety of network tasks, such as unicast and multicast networking. In addition to providing network connectivity, ACE supplies dynamic linking capabilities that could be used as part of a solution to distributing components of CVE software (see section 2.4).
While ACE itself has no specifically CVE-related functionality, it is used by the Octopus software described below, among others.

Plexus, developed at Iowa State University, is a software system for handling message transmission in a dynamic, peer-to-peer network of hosts. It is designed for a wide range of purposes, ranging from p2p file-sharing networks to CVEs. It is used by the sample implementation of the Datura software described in Chapter 4.

Plexus' key features are cross-platform message translation and the use of plug-in routing algorithms. The high-level routing algorithms in a network of Plexus sites can sit on top of a unicast or multicast (or mixed) network, transparently using the most effective communication channels available for a given message. Plexus provides an easy way to create a p2p network of sites with arbitrary connectivity between individual sites.

3.1.2. Distributed object technologies

A number of technologies for distributed objects have been created over the past decade, most famously CORBA (Common Object Request Broker Architecture [OMG02]) and DCOM (Distributed Common Object Model [Microsoft96]). In general, these technologies work by creating proxy shells of software objects on remote hosts. Method calls to these shells are transparently turned into network messages to the host of the "real" object. In some sense, these technologies represent an object-oriented variation of remote procedure calls [Srinivasan95], particularly in the means by which method invocations are translated back and forth to network messages.

Technologies such as CORBA were designed for general purpose network computing, and not explicitly for use in virtual environments. While they can be adapted to that purpose, they are usually found to be too heavyweight - too resource-intensive, and too high in latency. Also, the concept of converting every method call to a network request is frequently overkill for CVE applications, where it may be better to share a small set of common data that can (in some cases) be acted on locally.
Finally, these distributed object technologies usually rely on a client-server model, where one instance of a shared object, at a single site, is the "real" object. This means that the proxy objects on remote hosts can't themselves apply intelligent behavior - all processing and state changes are performed by the master. The proxies simply mirror the master's state. In general, the master copy of an object is not expected to migrate from one machine to another.

One particular application of CORBA technology to CVEs is Spin-3D [Picard+01], which used a multicasting variant of CORBA message passing to maintain synchronization for a set of identical objects on multiple sites.

3.1.3. NPSNET, HLA and VRTP

NPSNET is a series of CVE software tools developed by the NPSNET group at the Naval Postgraduate School [NPSNET]. This group's emphasis is on the creation of a network infrastructure that can support large-scale, persistent VR worlds. Their military background is demonstrated by interest in network protocols such as High-Level Architecture (HLA), designed for sharing information about military units in collaborative settings. A more recent related project is VRTP [VRTP99], which notionally provides a service for virtual environments similar to the one HTTP plays for web pages.

NPSNET projects have been particularly concerned with the dynamic requirements of large-scale CVEs. They suggest that in a virtual world akin to Gibson's cyberspace [Gibson84] or Stephenson's metaverse [Stephenson92], the environment as a whole must be available around the clock - that it would be impossible to perform a "global reboot" to perform maintenance or upgrade protocols.

The NPSNET group has investigated techniques for dynamically loading software modules to add functionality to a CVE world, including the ability to dynamically upgrade the communications protocols. One of the projects that came out of this work was Bamboo [Watsen/Zyda98], a framework for creating discrete software components in multiple languages which can be combined
together at application runtime. However, the most recent iteration of this work - NPSNET-V [Capps+00] - takes advantage of the dynamically loadable classes that are an inherent part of the Java programming environment.

Handling different versions of a component with different protocols is handled simply by tagging each message sent into the network with a protocol ID as well as an identifier for the component itself. Individual components can be upgraded without affecting the rest of the CVE environment, so long as this "meta-protocol" remains unchanged.

In order to actually support the large-scale virtual worlds they need on limited modern hardware, the NPSNET projects have taken various approaches to area of interest management [Macedonia95]. In keeping with the dynamically extensible nature of NPSNET, and the application-specific nature of many Aol techniques, their most recent environments use an extensible area of interest management system, where different "worlds" can define interest in terms of users, geography, explicit controls, etc. [Abrams99].

Finally, NPSNET has made use of the Lightweight Directory Access Protocol (LDAP) in order to provide naming and lookup services for its components [Howes+99]. LDAP is a popular, widely available protocol which provides unique names for the various components that make up a virtual world. The directory lookup capabilities provide a way to find the location (including host address, file name, URL, etc.) of a particular project, and is seen by the NPSNET developers as one of the keys to providing stable, persistent virtual worlds.

3.2. CVE libraries with simple entity management

Most CVE libraries provide some degree of high-level support for entity management. For example, they can handle the simple case of a user in one VE picking up a visual entity such as a chair and moving it around. The CVE software needs to perform a certain amount of arbitration - e.g. if two users try to grab the same entity simultaneously - and also needs to provide all the remote users with a reasonable graphical depiction of the user's manipulation of the object.
In this section, we discuss a few such libraries and the level of support they provide for entity management and manipulation.

3.2.1. DSO

DSO is a library for managing distributed shared objects. It is an Iowa State University project, and rests on the Plexus foundation described above.

Essentially, DSO provides a way to share any C++ object between sites in a networked environment. It allows any given site to lock such an object and modify its values, and then automatically updates those values in every other site in the collaborative environment. DSO's primary strength is that it can be used to add network sharing to almost any object with very little additional effort on the part of the application programmer - the application itself can be very naive about the network updating that occurs under the covers. Furthermore, DSO makes very few demands on the way an application has to be structured, beyond the basic assumption that C++ objects contain the information that needs to be shared between sites.

In terms of CVE-specific functionality, DSO stands midway between the projects discussed in the previous section and those that follow. It is tightly focused on this goal of object sharing, and does not itself provide features such as user avatars and communications channels, direct user manipulation of entities in VEs, and so forth. These capabilities can be added by the application itself or by a higher-level library that uses DSO/Plexus as its underlying network technology.

3.2.2. CAVERNsoft

CAVERNsoft [Cavernsoft] began as a research project at the University of Illinois at Chicago, and was subsequently released freely - but only for noncommercial use. It is composed of two somewhat-distinct parts. The first of these is a low-level cross-platform networking library, which includes system abstractions such as threading and synchronization primitives. It fills a role similar to the networking toolkits described in the previous section.

The second part is a higher-level toolkit which meets the sorts of specific requirements we
have discussed for CVEs. For example, this level includes modules for avatar representations, audio streaming, and entity manipulation. It even includes menuing objects and collision/navigation tools, which are useful in virtual environments regardless of whether or not collaboration is involved. The capabilities provided by these modules also tie CAVERNsoft to specific supporting software, most notably CAVElib and the OpenGL Performer [Rohlf94] scene graph library.

Actually, CAVERNsoft does not support manipulation of entities in an abstract sense. Rather, it supports picking and moving by directly modifying an object's visual representation, as provided by the Performer graphics library. While this allows applications to support users moving entities with very little programmer effort, it does not support higher-level manipulation (for example, turning on a virtual robot, or distributing that robot's autonomous actions and internal state). Supporting these kinds of behaviors requires programmers to deal directly with the networking level.

3.2.3. DIVE

DIVE [Frécon/Stenius98], or Distributed Interactive Virtual Environment, is a research project at the Swedish Institute of Computer Science [SICS02]. It provides a p2p networking interface over IP multicast.

DIVE does not inherently provide the sorts of high-level object controls we are hoping to evaluate. Instead, it takes an approach similar to a distributed database. In earlier versions, the database was fully replicated on all sites in the system. Later versions allowed individual sites to specify the subset of the database they were interested in, which would improve scalability (this is an example of area-of-interest management).

This particular system is interesting to us because certain DIVE applications introduced ideas of awareness between objects based on geographic proximity [Benford93]. This concept was adopted and expanded upon by the Massive CVE software, and will be discussed in more detail in a later section.
3.2.4. DOVRE

DOVRE [Hagen99] is a private project run by the Norwegian company Telenor. It is a highly-portable CVE library written to take advantage of ATM networks. It provides support for objects in a p2p networking structure. Every shared object in the environment has a single master instance and multiple slaves. The master copy exists on whichever node introduced that object to the simulation, and can not be moved. The slave copies simply replicate the (externally visible) portions of the master's state, and forward manipulation requests from users to the master copy.

In recent years, DOVRE appears to have been absorbed into a more general platform for distributed multimedia, with an emphasis on supporting MPEG-4 streams and desktop 3D, and less emphasis on the needs of immersive CVEs.

3.3. Advanced entity manipulation

The systems described in this section provide more powerful or customizable control over shared entities than those described previously. Some of them support migration of entities from one site to another (though only under a limited set of circumstances), and some provide persistence of entities after the site that introduced them to the virtual environment is removed.

3.3.1. Massive

Massive is the name given to a series of CVE systems developed at the University of Nottingham. Beginning in 1996, each generation of Massive has added significantly to the approaches it takes to CVE communications.

Massive-1 [Greenhalgh97] began as a prototype for exploring the spatial model of awareness [Benford93], which is one method for managing areas of interest between objects in the environment and determining when particular entities interact. In this model, every entity has two "auras", or spatial regions, which move with it in the virtual environment.

The first of these aurals is called "focus". Focus defines an area of space that the object
actively perceives. For example, the focus associated with a human user can be a cone representing his field of view. Massive-1 can handle multiple auras associated with different media. Thus, the visual focus of a user could be represented by a cone, while the aural focus is a sphere centered on the user. Of course, many entities in the environment will be inactive and unaware, such as the furnishings in a virtual room. These entities would have no focus region.

The other aura is called "nimbus", and it is the complement to focus. Nimbus is the region surrounding an object where it can be perceived. For example, the aural nimbus around a user could represent the region in which the user's speech should be audible.

Massive-1 included support for entities that modified the auras of other entities. For example, a megaphone could increase the aural nimbus of a user speaking into it.

While Massive-1 featured sophisticated management of awareness, it was never intended as a general-purpose CVE system. Rather, it concentrated on meeting the needs of virtual teleconferencing applications. For example, it does not have direct support for manipulating objects - a key concern in other areas such as design review or training applications.

Massive-2 [Greenhalgh97] elaborates on Massive-1's awareness policies with the idea of "third-party objects". These objects can manipulate the normal spatial awareness between entities or groups of entities. For example, a third-party object could expressly provide high-quality audio to all users within a group, or serve as a single proxy for a large group of entities (e.g. an icon representing a distant unit of individual soldiers).

In addition to this feature, Massive-2 was designed as a more generalized CVE system, with a well-defined interface for application development. It also supports a wider array of activities, including basic manipulation of objects in the VE.

Massive-2 also represented a move to multicast networking. By coupling the communications between particular objects to specific multicast channels, Massive-2 was able to reduce the overall network bandwidth requirements of the CVE.
More recently, [Purbrick/Greenhalgh02] presents recent work built on the framework of Massive-3. In Massive-3, individual updates to the states of objects are presented by specific events that are transmitted and received in a client-server network. Research in "deep behaviors" explores ways to apply filters and manipulations to these events as they transport through the system in order to achieve a number of effects.

This research allows event filters to be chained together in an arbitrary fashion. Individual events travel through the graph of filters, using pattern-matching techniques to determine the individual route based on various properties of the event. Individual filters can modify or delete events in whatever manner they choose.

The best way to clarify the purpose of these filters is to give a few examples: the LocalRouting filter sends the event to be acted upon locally, while the Unicast and Multicast filters send the event out on the network. The UpdateSceneGraph filter analyzes events to see if they cause any change to the local scene graph. Other filters provide total ordering of events and server side persistence.

These simple filters can be combined to create sophisticated behaviors. For example, the event ordering and persistence filters can be combined to provide trusted server-side persistence. A more elaborate sample is the Variant deep behavior, which uses filters to control when local modifications to an entity's state are forwarded to the server. The result is that individual users can manipulate and see their own version of a particular shared item, while another authority can examine the possible changes and decide which ones to apply to the master copy. This technique can be used to control the rate of change in the environment, or to prevent "virtual vandalism". [Purbrick/Greenhalgh02] provides several other examples of deep behaviors that can emerge from combinations of filters.

3.3.2. Deva

The Deva system [Pettifer+00] [AIG04] is a product of the Advanced Interfaces Group at
Manchester University. It applies a variety of innovative techniques in order to support very "large" interactive environments - those with a large number of users and/or a large number of complex, possibly active, entities. In fact, Deva is not just for collaborative use: its design applies equally well to running a very complex single-user application on a closely-connected cluster of computing machines.

At a high level, Deva appears to provide a client-server interface. A persistent server creates and controls all the entities in the environment, arbitrates access to them, and sends out updates about their changes. Clients - such as end-user visualization applications running on desktop computers or immersive VR systems - echo the state of the server and provide user interaction to the server.

The reality of the situation is more subtle: Deva uses a multi-level indirection scheme which replaces the single logical server with a (dynamically changeable) cluster of "server nodes". Each server node is responsible for a subset of the entities in the environment. Clients that want information about a particular entity send requests to the responsible server node. In theory, objects can be migrated from one server node to another (although this does not appear to be fully active in the current implementation). In order for the clients to find an entity after it migrates, each entity has a particular server node that is responsible for knowing its current location. Any client can determine the location of this "name lookup" server by performing a hash function on the entity's unique name. Finally, there is a mapping from the "virtual server names" (which are returned by the name hash) to the actual server node's location. This mapping allows individual server nodes to be added, removed, or replaced.

While not strictly required, Deva's protocols assume that the individual server nodes are geographically close and can communicate at high speeds and low latency. This emphasizes the logical appearance of a client-server architecture, where only the clients are located at significant geographic (or network) distances.
The internal state of the logical Deva server (and the actual server nodes) constitutes the objective reality of the virtual environment. The clients, however, see a subjective view that will more or less closely mimic this reality. For every entity "object" contained in the server, a given client will have a "subject" which mirrors the state of that object. The subject provides a way to depict the entity in the virtual environment presented by the client, and it provides a way for the client to control and manipulate the entity.

A trivial subject simply reflects the state of its object; with updated information sent to it whenever the object chooses to do so. More complicated subjects can behave in ways that provide a better end-user experience by less closely mimicking the object's state. For example, a subject might smoothly animate between two position updates from the object, or it might respond directly to a user's attempts at manipulation (correcting itself as necessary with the definitive reaction of the object located at the server).

The software representation of an entity in Deva is composed of "components" which provide particular behaviors, such as movement, boundary-checking, or the ability to be grabbed by a user. Components can be added or removed from an entity (its object and all its subjects) dynamically.

Each component has object and subject implementations that are used on the server and client sides of the system, respectively. Deva includes a library of components implemented in C++, and provides for the addition of new components by applications.

A particularly interesting aspect of Deva is that some of the components applied to an entity are determined by the entity itself (for example, an entity can determine if it can be grabbed by users), while other components are imbued by the environment. For example, an environment can determine whether it has gravity or not, and apply the components to implement that behavior to all the entities within it.

In short, while Deva is fundamentally a client-server architecture, it has many aspects that are applicable to purely peer-to-peer networking designs, particularly in the relations between its multiple
servers. The major architectural limitation, compared to a p2p architecture, is in the second-class nature of its "subjects" of entities, which are unable to participate fully in the CVE's computations. The idea of eliminating this subject/object and client/server dichotomy is one of the significant influences on the research presented later in this document. It is also not clear that all the difficulties of migrating active, interacting entities have been addressed in the Deva implementation to date – something which is an explicit goal of the Datura research project.

3.3.3. Continuum

The Continuum [Tran+02] project is a recent CVE middleware system developed by researchers at France Telecom. It is centered around the idea of visual objects represented as shared entities, where each entity defines certain “behaviors” that apply to it. For example, an entity using the "KinematicObject" interface has the properties of position and velocity, while a "Collidable" entity can check for collisions between itself and other Collidables. A class can inherit from several of these interfaces - for example, to create a moving object with collision detection.

Continuum is essentially a peer-to-peer network, although individual entities are handled in a client-server fashion. In Continuum, distributed entities have a "master" instance that typically contains the "intelligence" of the entity. The master instance is initially located on whatever site of the system first introduces the entity, but it is possible for the "master" quality to be transferred to another site if the object moves into a new geographic region of the simulation.

Replicas of the master instance can be created at other sites as necessary (generally, geographic partitions of the virtual environment are used to determine what set of entities a particular site needs replicas of). These replicas are slaves, and in a naive implementation they simply shadow the state of the master entity, which sends updated information periodically. Continuum, like Deva, also allows particular implementations to be smarter and perform dead reckoning or interpolation between updates. An intelligent master instance could also transmit velocity and acceleration data, reducing the need for frequent position updates. This can reduce both the amount of network
bandwidth used and the perceived network latency - since the resulting intelligent entity will appear to move smoothly on each host, barring possible error correction when a new position/velocity message is sent.

Key to Continuum's design is the ability to create new entity types suited to a specific application. While it is possible to create a basic set of classes to handle the features of many simple applications - for example, to handle passive objects that can be manipulated by the users of the environment - more complex applications may have unique requirements. This leads once again to the problem of needing to have a copy of the entity's class' code on every site of the virtual system in order to instantiate the needed slave replicas.

Continuum is implemented in Java, which has several powerful concepts which work together to resolve this problem. In addition to the cross-platform nature of Java bytecodes, it is also trivial to transmit an archive of Java classes to another site - which can then immediately use and instantiate those classes. So long as the site that creates the master replica of a Continuum shared entity has access to the necessary class files, it can transmit those classes to any site that needs to create a slave replica.

This implementation choice does have some issues which counter its benefits – the quality of Java implementations varies from platform to platform, and there has been little research in using Java solutions for fully immersive VR systems. The Continuum authors have used an Object Definition Language for defining their shared object classes which, they suggest, could be used to implement entity code in other languages with native compilation. However, it appears that this capability has not actually been pursued to date, and in any case this would eliminate the Java advantages previously discussed.

Finally, we should take a moment to discuss the methods for AoI management in Continuum. In the published examples of Continuum applications, the space of the virtual environment has been partitioned geographically, usually into a three-dimensional grid of cells. Each cell is put under
control of a single site of the simulation. When an entity enters a new cell, its master status is transferred to the entity at the site running that cell. That site is also able to share information about all its entities to other sites (e.g. to handle interactions that occur near or across the borders between cells). Individual entities can have aura properties (such as radius of influence or radius of perception) to determine what set of entities they interact with.

3.4. Gaming technologies

Since the release of Id Software's *Doom* [id93], interest in real-time online games has soared. Unfortunately, the nature of the gaming industry makes it difficult to provide a detailed analysis of technology. Networking technologies used in games are rarely discussed, either because of the competitive advantage gained from exclusive proprietary techniques or simply because of a lack of time to publish and present the results of research. Formal papers discussing techniques developed from game projects are still rare, and the researcher is left to cull information from less-formal discussions, magazine articles, or the actual source code released for older games. The result resembles forensic archeology more than computer science.

Many recent games have found customizability and expandability to be the keys to a long life and long-term profits. This became particularly prominent with the release of *Doom* in 1993. Fans of the game quickly developed new levels to explore. Many games followed suit by including "level editors" with the games themselves, and eventually this sort of expandability became a major selling point (see, for example, print ads for the game *Neverwinter Nights*[Bioware02]).

Of course, this creates the issue of how to bring these new data sets to the users - a problem many extensible CVE applications must consider (see section 2.4). The initial solution was to have the users manually download and install the new data. Of course, this proved cumbersome - especially when each player in a large game wished to have a customized model for his or her character. Later games would automatically install new models, levels, or data files, downloading them from the server running the game.
Downloading data files is relatively simple, but games did not stop there. Many "mods" (as they came to be known) included code as well as data. With this ability, fans of a game could create entirely new modes of play, and simple "shoot 'em ups" gave way to team play with complex objectives. This also introduced new problems and risks. For example, portability becomes a problem when trying to create code components for multiple platforms. If the code has to be compiled at each site, the entire problem becomes cumbersome and fraught with technical difficulties. And if the code is distributed as binary data, there are risks of virus-infected or Trojan horse code components. Unfortunately, games have typically chosen expediency and left these problems unsolved. Cross platform support for games in 2004 remains largely an afterthought. Those that have attempted to provide cross-platform expansion modules have typically relied on scripting languages or virtual machine languages like Java [Huebner00]. Others have attempted, for performance among other reasons, to abandon scripting in favor of binary code modules (e.g. the move from the "Quake C" scripting language in Id's *Quake* to binary mods in *Quake II* - followed by a flirtation with Java and then a return to ANSI C for *Quake III Arena* [Kreimeier99]).

The concern over distribution of binary code modules is just one of the peculiar security issues that networked gaming applications have brought to the fore. In fact, the common security concerns in games are quite different from those considered in academic CVE research. In business applications, for example, securing the data stream of an application from passive observers is often a primary goal to ensure the safety of proprietary data. This is seldom a concern in games, although the possibility of an observer disrupting or injecting into the data stream is occasionally considered. Primarily, though, game security is focused on the actions of the actual users of the system, and the ways they might find to cheat.

Cheating can take many forms, especially when users can add software modules via the game's expansion features. It could include the ability to see around corners, or computer assisted aim, or erasing damage to a character [Pritchard00]. For these reasons, many games (and gamers)
have preferred to use a dedicated, hopefully trustworthy, server to which all players connect. The server can keep control over state information and arbitrate the actions of users (e.g. checking if a player is shooting too often or too accurately to be unassisted). This leaves the issue of "information cheats" such as seeing around corners. These issues are more difficult to address, though there have been some attempts to ensure that users are running only "approved" unmodified game binaries [Netrek03].

Fairness is paramount to user satisfaction in games; in addition to preventing cheating, the game must also give the appearance of impartiality. Users on slow connections may experience less responsiveness than those on a high-speed LAN. Network traffic over the Internet is subject to various irregularities (bandwidth crunches, lost packets, irregular delays) that can frustrate users and limit their engagement. It can also lead to the impression that some users are suffering these effects to a disproportionate degree. Because of this, issues of network latency and "jitter" - the variability in latency - have become frequent grist for the complaints of gamers. Many games have gone to extravagant lengths to attempt to ensure that the actions of players occur in a realistic timeframe with clearly-evident cause and effect. For example, [Bernier01] gives an in-depth examination of how the timing of events was manipulated to improve the user experience in the game Half-Life [Valve98].

3.5 Octopus: In-house previous work

Finally, we consider a previous CVE project with which the author was closely involved. Octopus was a project which underwent several generations at Iowa State University's Virtual Reality Applications Center; the lessons learned designing and developing Octopus provided the most direct inspiration for the research discussed in this document.

The original version of Octopus [Lundin02] was a simple library designed to add collaboration support to VR applications using the CAVE library. It was implemented using ACE (see section 3.1.1) to supply its networking, and supported either unicast or multicast communications. It required a static network setup for a particular collaborative session, since each
node required *a priori* knowledge of all other nodes that would join the session.

In addition to simple graphical avatars of users (Figure 3.1 left), Octopus could distribute the position of each user's head and hand. It also distributed a set of boolean operators, which were used to share the state of devices such as the wands and joysticks which are common in VR systems.

This incarnation of Octopus did not have any direct support for entity manipulation. Instead, it relied on the notion that the same application code existed at each site and that the state associated with each user could be almost immediately updated - and that, therefore, the application would behave essentially the same on each site of the system.

This system worked well when all the sites were closely connected and communications were fast and reliable. However, this was not the case in real-world situations. When sites were widely separated, latencies would increase and the instances of the applications would fail to mimic each other correctly. For example, a user would grab an object locally, but on a remote site that user's hand position updated too slowly - jumping from one side of the object to the other without detecting the collision in between.

Based on these findings, a second generation of development occurred, detailed in [Hartling01] and [Hartling+01]. This work added explicit support for updating the state of arbitrary entities and managing ownership of entities.

![Figure 3.1. Development of avatars. Left: Octopus 1.0. Center: Octopus 2.0. Right: vjAvatar package, as used in Datura.](image)
In Octopus 2.0, every entity had a site assigned as its control server. The server would arbitrate which site could control the entity at any given time. Instead of simply assuming that control could be taken, as Octopus 1.0 did, in the new library a site would explicitly request control from the entity's server. The server would inform all sites about changes in an entity's control state, thus ensuring that proper synchronization was maintained.

Octopus 2.0 also allowed entities to share additional state information - for example, a virtual robot could share information about the status of its various indicator lights, or send out position updates based on its self-controlled movement. Some Octopus applications, such as the Robot Playground [Just00], even experimented with changing the location where an entity's internal computations occurred. In the Playground example, robots would choose to perform their AI calculations at the site of whichever user pushed their "on" button.

Octopus 2.0 included somewhat more elaborate avatar support (Figure 3.1, center). Avatars were treated as just another example of shared objects, and used the same mechanism to share their state. The only difference was that avatars were owned automatically and permanently by the site that introduced them into the simulation.

While these enhancements allowed for the creation of much more complex and robust collaborative applications, they had significant shortcomings. Octopus 2.0 required a considerable amount of support code in applications that used it - for example, all the code for requesting control of an object and moving it was located in application space, with only the handling of the request and response messages occurring in the Octopus library itself.

The implementation of the message-passing system proved more troublesome. Messages were constructed as C++ structures, and their in-memory representation was passed directly from one site to another. This implementation required significant abuse of C++'s RTTI (run-time type identification), and essentially prevented any interoperability between Octopus applications running on heterogeneous computing platforms.
The connection handling and message interoperability limitations of Octopus 2.0 formed part of the inspiration for the Plexus software discussed above, and Plexus in turn forms part of the underlying technology of the new work described later. The crude ability to manage where control of an entity was centered inspired much of the thinking on location of computation described in chapters 5 through 9.

3.6 CVE requirements revisited

Chapter 2 concluded with a summary table of the technological issues involved with creating CVE applications. Table 3.1 (page 50) extends that table to show the correspondences between those issues and the information described in this chapter. It briefly describes the current approaches taken in each major area of requirements, and notes those that particularly require additional research.
<table>
<thead>
<tr>
<th>Networking  (addressed by all)</th>
<th>Several (e.g. ACE, some object protocols) use multicast communications.</th>
<th>Plexus explicitly focuses on p2p network connection issues.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Distributing entity state  (addressed by all CVE-specific toolkits, but not directly by networking tools in 3.1.1, 3.1.2)</td>
<td>Most toolkits address in some way.</td>
<td>Several (e.g. Octopus, DIVE, NSFNET) use multicast p2p updates.</td>
</tr>
<tr>
<td></td>
<td>Updates sent when data changes or periodically.</td>
<td>Dead reckoning and other interpolation techniques common.</td>
</tr>
<tr>
<td>Area of Interest management (addressed by Massive, DIVE, Continuum, etc.)</td>
<td>Well-established techniques based on spatial representations or geographic partitions.</td>
<td></td>
</tr>
<tr>
<td>Entity Control and ownership (addressed by all CVE-specific tools)</td>
<td>Server-based solutions are common.</td>
<td>Some (Deva, Continuum) allow limited/partial migration of control. Others (Dovre, Octopus, etc.) do not.</td>
</tr>
<tr>
<td>Creating environment (addressed by all CVE-specific tools)</td>
<td>Server-based solutions are common.</td>
<td>Few techniques for p2p networks, mostly based on predefined geometric partitions of space (e.g. Continuum).</td>
</tr>
<tr>
<td>Persistence (usually handled at application level, but see e.g. Massive, NPSNET)</td>
<td>Work on world storage formats ongoing.</td>
<td>No clear solutions to versioning without a central server/repository.</td>
</tr>
<tr>
<td>Location of Computation (addressed directly by Deva and Continuum, implicitly by others)</td>
<td>Many server-centric solutions.</td>
<td>Some multi-server solutions (Deva, Continuum), with limited support for migration or limited circumstances where migration is used.</td>
</tr>
<tr>
<td></td>
<td>No general approach to handling location of computation using a variety of factors in a p2p environment.</td>
<td></td>
</tr>
<tr>
<td>User communication (addressed by all CVE-specific tools)</td>
<td>Large variety of avatar techniques (see section 2.8).</td>
<td></td>
</tr>
<tr>
<td>Presentation (addressed by CVE-specific tools)</td>
<td>Many web-based interfaces (e.g. many MASSIVE applications; see also section 1.2.1).</td>
<td>Immersive interfaces using established VR libraries (e.g. Octopus w/ VR Juggler, Deva w/ Maverick).</td>
</tr>
</tbody>
</table>

Table 3.1. A summary of how previous research has addressed the requirements of CVE software; compare to table 2.1 for further background. Of particular interest is the reliance on server-based (or multiple-server-based) solutions to entity management.
4. Datura: A framework for investigating Location of Computation

This chapter begins the description of this dissertation's new research. It introduces Datura, a test framework for entities that distribute their computations across the sites of a networked virtual reality system. It describes how Datura distributed objects are represented, and how the networking and synchronization can be achieved. The goal is to provide the necessary background to understand Datura's structures and algorithms. Datura will be used to give a context and concrete examples for controlling LoC. In the following chapters, we shall discuss the development process and experiments used to create Datura's advanced capabilities for managing entity state and controlling the location of computations for entities. Those capabilities are the core of the new work presented herein.

We should begin with a statement of philosophy. The goal here is not simply to replicate the state of an object throughout the network (for which many generic technologies exist). Rather, the goal is to create, for the users of a networked virtual environment, the illusion that they are interacting in a shared space. This distinction has interesting consequences.

A Datura entity is crafted to provide the appearance that all of its state is instantaneously replicated anywhere in the system. To do this, it can decide how much of its internal data actually has to be updated, and how often. It can decide how much of that state can be calculated independently on multiple hosts. It can communicate data at the highest possible level, and synchronize itself as rarely as possible. It can decide which sites to perform its key calculations at, and what impact that will have on the users and their perceptions. It can provide radically different depictions of itself to different users if it can expect to get away with it, without breaking the sense of immersion. It can be as "smart" as its programmer's skill and imagination can make it. It can cheat, or it can cheat outrageously.
A few examples can illustrate our intention. If an object is moving in the environment, its entity might choose to move independently on separate sites, synchronizing the copies of itself only periodically and allowing the individual copies to extrapolate between updates. It might send "higher-level" information than simply position updates; it might send a path for other sites to follow, or just send a destination and the time allowed to get there. Objects that a user interacts with might appear to respond immediately, but they might not inform other sites of those interactions until (unless) they receive permission from some authorizing agent.

4.1. The structure of omnipresent entities

The most fundamental concept in Datura is the omnipresent entity. Everything in a Datura application that needs to be represented on multiple sites of a CVE is created as an omnipresent entity. A particular entity might represent a user's avatar, or a piece of terrain or furniture, or a building block, or the virtual representation of an autonomous agent.

In order to provide this kind of flexibility, the omnipresent entity in Datura is essentially an empty vessel. It has a GUID [TOG97] which is used to uniquely identify it, and very little else. Properties and capabilities are added to an entity by attaching elements to it - as demonstrated below.

Entities are omnipresent because, in a networked environment, there will be copies of each entity on each site of the network. If two entities on different sites have the same GUID, they are assumed to be mirrors - representations of the same shared object. Each mirror of the entity will have the same set of elements attached to it, and the mirrors of a particular element will communicate over the network to keep an appearance of shared state.

4.1.1. Elements

An element in Datura is a C++ object that can be attached to an omnipresent entity to imbue it with some kind of property. These properties are data storage, processing or intelligence capabilities, and graphical (and sonic, haptic, etc.) depictions. The categorization is similar to the three parts of the well-known Model-View-Controller (MVC) design pattern [Krasner88]. Perhaps
the best way to clarify the idea is to describe some of the elements that are part of Datura's basic library:

**Elements that add data storage.** The simplest elements are "receptors" - they don't actually perform any computation, but merely provide data elements that can be used by other elements. In the Model-View-Controller pattern, receptors would form the model.

- **PositionReceptor** - This receptor imbues an object with the quality of simple location. That is, it has a position, orientation, and scale in 3D space.
- **ModelInfoReceptor** - This receptor provides information about a model - for example, an FLT or VRML data file - that can provide a graphical depiction of an object.
- **SizeReceptor** - This receptor can store information about an object's current height, width, and depth. It is particularly useful for objects that can be dynamically resized or nonuniformly scaled.

**Elements that perform computations and activity.** These elements provide many sorts of activity - that is, they actually make an entity do something. Many of these elements can be thought of as the Controller part of an MVC model.

- **TrackedElement** - This element can be used to connect an object to a VR Juggler positional input device. TrackedElement writes data to its entity's PositionReceptor. Thus, the object will follow a particular tracker. This is particularly useful for representing user input, or creating avatars of users.
- **ManipulatorElement** - This element receives input from a user, and generates commands to interact with other entities. It is frequently combined with a TrackedElement to create the shared impression of a VR wand. The ManipulatorElement of a wand entity can talk to the ManipulatorTarget and GrabbableElement of, say, a chair entity, so that the user can pick up the
chair and move it around.

- **ManipulatorTarget** - This element is attached to entities that can be manipulated by a user. It determines whether or not to allow a manipulation, and connects manipulators with manipulation handlers (such as GrabbableElement or RemovableElement).

- **GrabbableElement** - This manipulation handler element attaches to any entity that a user should be able to move around. It does the actual work of making the entity move in accordance with the user's input.

- **MultiGrabbableElement** - This experimental element can be attached to an entity in place of the GrabbableElement. It fulfills the same role, but allows multiple users to grab an object and, in theory, position it cooperatively.

- **ResizeableElement** - Objects with this manipulation handler attached can be resized along any of their axes. ResizeableElement writes its results to the object's SizeReceptor.

- **RemovableElement** - This is another manipulation handler element that can be triggered by a ManipulatorElement. When it is activated, it will delete its entity on every site in the CVE.

- **ObjectCreatorElement** - This element is a special case, as there will always be one (and only one) entity on each site of the CVE that uses it. It is responsible for creating the local mirrors of omnipresent entities that exist elsewhere in the network.

**Elements that provide graphical depictions.** These elements provide graphical views of omnipresent entities - they are the View part of an MVC model. They can provide scene graph nodes or drawing routines that can be used by an application. Generally speaking, there are several versions
of each of these elements, specialized to support different scene graphs and graphics libraries. As far as the rest of Datura is concerned, the elements for different scene graphs are identical - so that (for example) an OpenGL Performer [Rohlf94] Datura application can communicate with an OpenSG based application.

These elements often need to load geometry of models or avatars, usually in the form of output files from a 3D modeling package. In the Datura sample implementation, these data files are expected to already exist on each participating site. Datura only distributes information needed to render them inside the virtual environment: the model and file name, current position and orientation, scale, etc. A future enhancement would be to transfer model data files automatically, using a file transfer protocol in parallel with Datura's current state distribution capabilities.

- **ViewableElement** - This element uses information from a ModelInfoReceptor and PositionReceptor to provide a simple view of an inanimate object. It is used for presenting chairs, boxes, terrain, etc.

- **AvatarElement** - This element provides an animated human figure via the vjAvatar library. It is typically used in an entity with a PositionReceptor and a TrackedElement (which are used to provide the position of the user's head). It can also interact with other entities that provide the positions for the user's hands or other points on the body.

4.1.2 Putting them together

As described above, a shared object in Datura is simply an omnipresent entity with a set of elements attached to it. Of course, there are many possible combinations. In this section, we'll look at a few common varieties and how they work together. For the moment, we will limit the description to environments with a single site, and explore the networking ramifications in later sections.
A manipulator for user input. Our first example (Figure 4.1) is a user-input device. This is the software representation of some physical input device, such as a wand with various control buttons. It might also represent a mouse, a glove, or any other device where we can interpret the inputs as a position in 3D space and a set of binary control values.

We construct this object by creating a new omnipresent entity and attaching three elements to it. First, we want to know the object's position in 3D space, so we add a TrackedElement (which attaches to a VR Juggler input device) and a PositionReceptor (which stores the values given from the TrackedElement). Next, we add a ManipulatorElement. The manipulator also attaches to a VR Juggler input device, but in this case it's a digital input instead of a positional input\(^5\).

The ManipulatorElement does most of the work associated with this object. When the user presses a particular button on his wand, the element becomes active. It then searches for other omnipresent entities that are intersecting with its position. If it finds one with the right element

\(^5\) There can be many kinds of manipulators, some of which can have more complex or versatile
attached (GrabbableElement, for example), it requests to be added as that element's current manipulator. When the user releases the button, it will release itself from any entity it is currently manipulating. We will explore the other side of this interaction in the next example.

At this stage, our input device has no element attached that would give it a visible presence in the virtual environment. If we chose to, we could add a visual to it the same way we will for the entity in the next example.

**An inanimate, manipulable object.** Our second example is the virtual representation of a simple object such as a chair. The object is inanimate by itself, but the user should be able to manipulate it in a variety of ways, such as by picking it up and moving it around, or removing it entirely from the simulation. Figure 4.2 illustrates such an omnipresent entity. We'll begin by considering the elements that are necessary to provide the visual representation of the chair within our environment.

![Diagram](image)

**Figure 4.2.** An inanimate object that can be grabbed by a user, with all its attendant Datura elements.

...activation mechanisms than a simple button press.
First, we have a PositionReceptor, just as with the previous example. The chair shouldn't move by itself, so it does not use a TrackedElement. The initial coordinates of the chair will be determined by the application when it creates the PositionReceptor.

Once we know where it should appear, we need to know what should appear there. This is where the ModelInfoReceptor comes in — it provides the name of a 3D geometry file and parameters such as the scale of the model. Next, a ViewableElement is added. Remember that we only distribute the file name and parameters for drawing the file, and not the geometry data itself. The viewable reads the data from the ModelInfoReceptor and the current location from the PositionReceptor and provides the application with a method or data structure that can be used to represent the entity in our application's drawing routine or scene graph.

Next, we need to add the elements that will allow our user to interact with the chair. First, there's the ManipulatorTarget, which controls access to the entity. We described in the previous example how a manipulator would attempt to attach itself to an entity. The ManipulatorTarget decides whether to accept this or not - it is in charge of synchronization if two requests arrive simultaneously from different manipulators, and may also perform some form of authorization check on the manipulator.

Finally, we need elements that will actually handle the manipulation. One of these is GrabbableElement. When a manipulator is attached to GrabbableElement, it periodically updates its entity's PositionReceptor so that the entity follows the manipulator's movements. The ViewableElement will then update itself based on that changing location, with the result that the object will appear to move in space with the user's hand. When the user releases his "grab" button, the manipulator is deactivated and the object stops moving.

---

6 The manipulator can actually have several buttons assigned to it, with each configured to interact with a different element in its target entity.
The **RemovableElement** is similar. When a manipulator is accepted, it immediately tells the manipulator to detach itself. It then signals for itself, its omnipresent entity, and all other elements attached to that entity to shut themselves down and remove themselves from the system. The visible result is that when the user presses his “remove” button, the touched object disappears from view (the **ViewableElement** will remove the entity's appearance from the scene graph as part of its shutdown routine).

**Keeping track of entities.** The interaction described above requires that a given omnipresent entity be able to find other entities, such as possible targets for manipulation. Datura maintains a repository that records all the entities that a particular site knows about. When a site creates an entity, it must be added to the repository in order for other entities to interact with it.

This is simple in the case of locally introduced entities, but we must also consider the discovery of remote entities in a networked environment, and how they are referenced in the repository (see section 4.2.1, below).

### 4.1.3 Updating elements

Most elements have an `update()` method, which is used for whatever computations it needs to perform. This method is where a manipulator element decides to request attachment to a **ManipulatorTarget**, and where a **GrabbableElement** or **TrackedElement** updates the coordinates in a **PositionReceptor**.

Datura provides a scheduler that is responsible for periodically calling the `update()` methods of every element. It assures that the element has been properly initialized before `update()` is ever called, and makes a few guarantees about synchronization and shutdown conditions for the element.

Of course, the element isn't required to perform all computations in `update()` - it could also spawn off a separate thread to perform time-consuming or continuous computations. In this case, the `update()` method becomes a convenient place to record temporary results so that they can be used
by other elements – for example, to update the graphical depiction of an entity or to answer queries from other elements.

4.2. Entities in the network

Most of the discussion above only considers how omnipresent entities are created on a single site. Of course, these entities and their associated algorithms are perfectly usable in a single-user, non-networked application, but the primary purpose of the design is to support networked applications. In this section, we will explore the basic issues of managing copies of Datura's omnipresent entities in a CVE – creating entities, advertising their presence, and distributing their representations. The next chapter will introduce the techniques used by various elements to maintain and update their state across the network.

4.2.1 Creation of local and remote entities

Every object in a distributed system has to be introduced somewhere. In Datura, this primordial copy of an omnipresent entity might be created on a particular site for any of a number of reasons. It might be created to represent one of the users local to that site, or because of a configuration file or option read by a particular instance of a shared application. It might also be created as the result of actions taken by another omnipresent entity -- one can imagine an entity with a "Creation" element that adds new objects to the environment as a response to user input.

The first instance of a particular entity is referred to as the primordial mirror. Whichever site creates this entity is responsible for assigning it a unique GUID and creating its initial set of elements, as described above. Finally, the newly-created entity is added to that site's repository.

There is nothing inherently special about the primordial mirror of an omnipresent entity. Of course, the application that creates it gets to decide its initial state, set any default values for its elements, and so forth. If one of its elements uses multiple states in order to assign authority or distribute computation, the primordial mirror will generally be created as a "master", as we shall see in Chapter 5.
There are a variety of ways that the other sites of a network can find out about such a newly-created entity. The originating site will send an announcement when it adds the primordial mirror to its repository. Alternately, when a new site attaches itself to a network it sends out a query, asking the other sites to send it information about the omnipresent entities which already exist. Sometimes, due to the timing vagaries and unreliability of network communications, a site might receive a message meant for one of the elements of an entity it has never heard of. As a robustness strategy, it can send out another query, asking the other sites for a description of this mysterious entity.

4.2.2 Creating mirrors and distributing initial state

When a site in the network learns about an omnipresent entity that was originally created elsewhere, it creates a local mirror of that entity. The local mirror is simply an instance of the Datura omnipresent entity class which is assigned the same GUID as the primordial mirror, and which has duplicates (also called mirrors) of all that entity's elements attached to it. Once created, the elements interact with the copies of themselves on other sites in an attempt to maintain a consistent shared state.

In order to facilitate the creation process, every site in a Datura network has a special ObjectCreator. The ObjectCreator sends out the queries and announcements described above, and responds to requests from other sites. When an ObjectCreator receives a request for information about a particular omnipresent entity (or about all known entities), it finds the entities in its local repository and creates a set of messages which capture their internal state.

The message data begins with the object's unique GUID, and is followed by a listing of the elements attached to it. The elements are also identified by GUIDs, although in this case the GUID is defined on a per-class basis, instead of per-object-instance. Ultimately, this means that any instance of an element in a Datura network can be identified by the element class GUID and the entity identifier GUID. Furthermore, the individual mirrors of a particular element can be distinguished by the name of the site they exist on.
Figure 4.3. Object creation. Top: Alice (at site A) and Bob (at site B) start their applications; each creates a user avatar and a chair to place in the CVE. Bottom: Site A and Site B connect; each creates mirrors of the entities introduced by the other site.

This information is followed by brief encapsulations of the current state of each element - essentially, each element class decides how much information it needs to send in order to get a new mirror of itself "off the ground." This could range from a complete description of the element's internal state (as is the case with ModelInfoReceptor), or it could be nothing at all (i.e. TrackedElement, which sends out periodic state messages to all of its mirrors anyway).

When the ObjectCreator on the site that requested information about an entity receives this message, it creates a new omnipresent entity with the specified GUID, adds the necessary elements, and then unserializes the initial state for each element recorded in the message. The new mirror is then added to the local repository and becomes live. If any viewable elements were attached to the entity, the repository notifies the application so that they can be displayed.

4.2.3. Substitution of element implementations

So far we've spoken of the mirrors of an element as being similar objects – in implementation
terms, all instances of the same class, uniquely identified by the GUID of the entity they attach to. This does not necessarily have to be the case - two mirrors of an element really only have to be the same insofar as they support the same interfaces for communicating with other elements and the same communications protocols (i.e. message structures) so that they can communicate with each other.

This allows us to perform some useful substitutions. The most obvious example is that the elements on two sites could be compiled for different architectures. In the sample implementation of Datura, the underlying network layer (Plexus and VPR) handles all necessary data translations, allowing, for example, a mirror on a big-endian site to communicate with a mirror on a little-endian site.

Substitutions can also be used to support different VR systems or graphics APIs on different sites. We've already mentioned the ViewableElement that provides a graphical depiction for a given omnipresent entity. In terms of implementations, there can be various versions of this element written to interact with a variety of scene graphs and graphics APIs. For example, an application on one site of a collaborative VR system may have its entities adorned with PfViewableElements (the ViewableElement that understands SGI's OpenGL Performer). A new site might join and create mirrors of all these elements, but actually create instances of OpenSGViewableElement instead. The mirrors are ignorant of this heterogeneity, but the result is that two applications running on different platforms with different available graphics libraries can talk to each other. One could even add a "dummy" version with no actual graphics code, and run the application on a site with no graphics resources at all (perhaps simply using it to perform computations for one or more complex elements).

The issue of actually creating these various versions of an element is handled via negotiation between the application (which knows what its graphics capabilities are) and Datura's element factory (which knows what implementations of a particular element are available).
4.3. Communication inside of elements

In the Datura element model, all network communication happens between the mirrors of a given element; communication between one element and another happens via local method calls. For example, there is no direct network message passing between a ManipulatorElement and a ManipulatorTarget. Instead, the manipulator's mirror on a particular site will make a method call on the local mirror of the target. If necessary, the target will send messages to its mirrors at other sites to decide whether to allow manipulation.

This communications model serves to separate the networking technology from the semantics of element interactions. In the Datura test implementation, communication is handled by the unicast/multicast message capabilities of the Plexus networking library. However, the implementation could be replaced with, for example, a distributed shared memory mechanism or an RPC implementation without affecting the element model described in this chapter. Such a replacement would alter the message-passing and role-management techniques that form the basis of the next chapter, but it should be remembered that these alternate networking technologies are simply implementations on top of message-passing protocols in the first place.

Chapter 5 examines how an element might manage data, synchronization, and message passing among its mirrors. As suggested in the introduction to this chapter, that can be a very subtle problem. It is very dependent on the job a particular element has to perform - what sort of internal state it has, or what computations it must perform, or what interactions it must have authority over. Adding to the problem, it may also depend on who is watching (or what other elements are looking at a given element's results). We will suggest several general strategies that an element might use, and explore in depth multiple examples of elements from real-world applications.

4.4. The entity/element model and location of computation

The research goal of this thesis is to control the location of computations that are performed
for the various elements that we implement. We must also decide how to use this control, and what circumstances will be considered for triggering the movement of computation and control from one site to another. This section revisits the ideas for using location of computation proposed in Chapter 4, and puts them into the context of an element-based CVE design such as Datura.

4.4.1. Control based on load balancing or stress

This is the most common use of task migration in distributed computing, in which tasks are distributed among a series of sites based on the processing load at each site [Murthy/Manimaran01]. This form of load balancing is most effective when dealing with a single task that can be divided into independent, computationally equivalent units. It is somewhat less suited to handling a large number of independent entities with radically different computational requirements.

4.4.2. Control based on location of concerned users

This aspect of computation location control puts the user experience foremost. The simplest explanation is that, ideally, computations should occur somewhere close to the users that are most directly observing or interacting with them. "Close" preferably means "on the same computer", but might also be defined in terms of the network latencies involved between different sites.

To see why this is advantageous, consider the example of a user picking up a virtual box. To give the best experience, the box should smoothly move with his hand movements. If the computations for the box's movement occur on another site, there will be a delay as the user's movements are transmitted and the box's responses are transmitted back. This can make it difficult to position the box precisely, and limits the user's sense of immersion in the environment.

Instead, when the user grabs the box, we could perform the computations for its behavior at the site hosting that user. Then, the box's reactions can occur immediately without the network lag – at least from that user's point of view.

It is also possible to "cheat" in various ways. We could keep the canonical control over the box's position at a remote site, but have the element provide approximate results on every site –
providing the appearance of immediate responsiveness to the user, and providing an appearance of smooth movement to other remote users who might be observing the interaction.

4.4.3 Control to increase robustness; dealing with network disconnections or dying sites

As CVEs grow larger in scale, they also need to become more robust and dynamic. If there are only two sites, then it makes little difference if the application needs to be restarted if one crashes. If there are fifty sites, this becomes untenable. Our CVE applications must be able to deal with sites leaving in the middle of a session, even if that departure is unannounced - the result of a network failure or software crash.

Therefore our elements must deal with the notion that any site - and all the state information retained at that site - might suddenly vanish without a trace. This is particularly important when elements put a single site in charge of making "canonical" decisions, such as deciding which of two users grabbed something first.

In the case of a grabbing element, it may be possible for the element to simply detect that an important site has disappeared and elect a new site to handle the control. This might occur rarely enough that the minimal disruption - forcing the entity to be dropped and picked up again, for example - would be acceptable.

Unfortunately, this is not always the case, especially with highly intelligent elements that store a considerable amount of state. Consider a robot behavior element that implements a pathfinding algorithm. It might be unacceptable to force the robot to rebuild its geometry database and paths from scratch when a controlling site dies. At the same time, replicating all of that state information at every site in case of a crash is wasteful and unduly paranoid. In designing our elements and the CVE software to support them, we should look for opportunities to take the middle road - finding a balance between replicating state information for robustness and minimizing the network traffic required for the system and its behaviors.
5. Distribution and communication patterns of elements

In order to devise methods for controlling Location of Computation in CVEs, we must first understand how computations can be distributed among multiple participating sites. This chapter describes several general strategies for breaking up the computations and decision-making capabilities of individual elements, which we will refer to as distribution patterns.

The second half of the chapter categorizes many of the elements that are likely to be found in real-world CVE applications. Based on the functionality and interaction that these elements provide, we group them in broad categories called interaction patterns. We also discuss which distribution patterns are particularly suited for various kinds of interaction.

Subsequent chapters will build on this framework, describing methods for handling the various distribution patterns in networked systems such as Datura, and exploring how to manage location of computation and perform transfers of computation between sites.

5.1. Distribution patterns

A distribution pattern defines how computations and decision-making are divided among the mirrors of a particular element. For example, in the model described in the previous chapter, a particular omnipresent entity – "the red chair", for example – will have a number of elements attached to it. These elements (grabbable, removable, resizable, etc.) will have mirrors of themselves on each site collaborating in the CVE. The distribution pattern applied to the GrabbableElement will decide how all the individual mirrors of the red chair's GrabbableElement will interact with each other.

The distribution pattern does not directly influence how the mirrors will interact with other elements, though it may impact exactly what capabilities the element has.

Every mirror of an element will have a particular role assigned to it. The available roles are defined by the distribution pattern the element uses. Some of the roles we will explore below include
"Peer", "Slave", "Master", and "Processor". Much of the location management work in subsequent chapters will involve deciding which mirrors assume which roles, and when to change the role of one or more mirrors.

We describe three broad categories of distribution, named after the most important roles each uses. Following this we describe a few variations, special cases, and possible expansions of the initial types.

5.7.7. Peer/Peer distribution pattern

The simplest distribution pattern simply assumes that all mirrors are equals — peers — with no distinguishing authority. In the Peer/Peer (P/P) pattern, all the mirrors of an element function independently. This means that there is no central decision-making or synchronizing authority.

P/P distribution is appropriate when the results or activities of the element can be independently calculated on each site. Viewable elements, where the element creates a visual representation of an entity based on the status of its other elements, are an example of an element where P/P distribution can be useful. Receptor elements are also P/P, since they simply provide data units that are read and written by other elements.

![Peer to peer distribution](image)

Figure 5.1. Peer to peer distribution. Every mirror of a P2P element is equal — they may freely exchange data between themselves and other mirrors, but no one mirror has any authority over any other.
In many cases, P/P elements will engage in little or no network traffic – the individual mirrors are completely independent. In other cases, they may share information between themselves, but only if no central data synchronization is necessary. For example, one could conceive of an element that gathers certain information at each site and shares it among its mirrors. In fact, part of Datura's location management is actually managed in just this way: Location Managers at different sites share information about available resources, request that work be offloaded to other sites, and so forth, but there is no central organizing authority among these manager elements.

5.1.2. Master/Slave distribution

Many common interactions involve relatively little computation, but do require controls for synchronization, authorization, and so on. One simple example is the ManipulatorTarget that is used for any object that can be picked up and moved around by a user. This element proxies for all the elements that do the actual work of manipulation, handling their synchronization and permission management.

The synchronization concerns are significant. First, the manipulation target needs to make sure that manipulation always begins and ends at the same positions on each site (so that the manipulated object is always released at the exact same position at all sites, resized to the exact same final dimensions, etc.). This can be complicated by irregularities in networking – the various mirrors may not all have the same information about the position of the user's hand at the exact same instant.

The second concern happens when two users decide to manipulate the same object at the same time. If each mirror decided the matter independently, it's possible that some mirrors would give the object to one user, and some to the other. In order for the GrabbableElement to work correctly, we must ensure that all mirrors accept the same decision.

The easiest way to do this is to give one mirror authority to make these decisions, and require all other mirrors to honor its results. In a Master/Slave (M/S) distribution, one mirror is given the role of Master, while all other mirrors are given the role of Slave. The master is responsible for all
decision making, and for updating the slaves with whatever information they need. Slave mirrors typically contact the master whenever they need an authoritative result, though they may be able to answer some queries locally.

In Datura, the primordial mirror of a Master/Slave element is initially given the Master role. Under certain circumstances, it may be necessary to transfer that role to a mirror at another site. For example, this could happen if the site hosting the master loses its network connection, or if it becomes overloaded and needs to transfer part of its workload to other sites. These circumstances, and methods for handling a transfer of the Master role and preserving all necessary state information and so forth, are discussed in the following chapters.

5.1.3. Master / Processing Group / Slave distribution

The third distribution pattern we introduce is an enhancement of the Master/Slave scheme, designed to better support elements that require significant amounts of processing. Such elements could include, for example, physics calculations, pathfinding or other AI tasks, data analysis, etc. They key element is that the computation is in some way suitable to distribution; i.e. that it can be broken into multiple pieces that can be addressed somewhat independently.

Figure 5.2. Master/Slave distribution pattern. Slaves may forward requests or queries to the master mirror (dashed line), but all decisions and definitive data originate from the master.
In the Master / Processing Group / Slave (M/PG/S) distribution pattern, the mirror taking on the Master role is once again responsible for final decision making and synchronization. The remaining mirrors are broken into two groups: the slaves, which take a purely passive role, and a processing group (PG) of mirrors that can assist the master with computations.

In this pattern, the master can break a computation into multiple pieces, each of which can be performed on a separate mirror. The master can then receive data from all the processor mirrors to create a complete solution.

As an example, consider a simple pathfinding task. The master decides when a pathfinding computation needs to occur, determines the desired start and end points, and communicates this information to the mirrors in its processing group. These mirrors can then perform calculations independently (perhaps using different techniques, or focusing on different parts of the search space, or using different randomization methods), and communicate results back to the master. The master can then decide which of the solutions to use and send commands back to all the other mirrors (processing group member and slave alike) to follow the chosen path.

Figure 5.3. In an M/PG/S arrangement, the master can offload processing requests to other sites and receive possible or partial answers back (dashed lines). The master is responsible for combining partial answers or choosing from available solutions.
The purpose of distinct slave and PG member sites is to allow load balancing and other forms of resource management. Recall that mirrors of a given element exist at all sites; this technique lets us control which sites will take on the load of a given element's computations. This distribution pattern adds a number of complexities to the handling of location of computation. Firstly, a set of PG mirrors needs to be selected and maintained. The size of this group could depend on a number of factors, including the number of available mirrors and their available computing resources, as well as the divisibility of the computation algorithm itself. As with the Master role, it may be necessary for a mirror to stop participating as a PG member or to transfer that role (possibly along with partially completed computations) to another site.

5.2. Variants of distribution patterns

The three distribution patterns described above can adequately describe a large number of elements. Below, we describe several variations of the basic patterns.

5.2.1. Multitiered processing groups

The M/PG/S distribution essentially partitions the set of mirrors into two groups: those that participate in computations and those that do not. Extending this idea, we could consider additional levels of participation, such as “medium-load” versus “high-load” members of the processing group. This would allow finer-grained control over load balancing than the basic scheme. For example, a PG mirror on a heavily-loaded site could simply reduce its level of participation rather than dropping to the slave role. This in turn would reduce the need to transfer roles from one mirror to another (which can be expensive, as we will see in the next three chapters).

This idea of multiple levels of participation can be conceptualized in a number of ways. For example, we might create multiple distinct roles (“High-load Processing Group”, “Low-load Processing Group”, etc.). This would require us to maintain a set of members for each group, but is well suited for elements that need to perform several distinct kinds of computation. Alternately, the load could be manipulated entirely by the master (by deciding what information each processor
receives) or entirely by the Location Manager at each site (e.g. by limiting the timeslices that a given mirror might receive for processing).

5.2.2. Backup sites

Sometimes, location management will force a given role to be abandoned by one mirror and adopted by another. For example, a site preparing to leave the CVE can no longer host a master mirror, and a site suffering heavy processor load may be unable to sustain as many PG mirrors as it previously agreed to.

Transferring such a role, along with whatever state information it includes, can be time-consuming. Doing so without disrupting the activities of the element can be more complicated. One way to address some of these issues is for a mirror to select another mirror at another site as its backup mirror. A backup mirror can be proactively updated by the master (or PG member), or it can duplicate the Master's computations. That way, when a transfer is necessary, the backup can be ready to take over at a moment's notice.

Another important role for a backup site is to provide redundancy in case of a system or network failure. If a site crashes or unexpectedly goes silent, a backup mirror can take over for the suddenly missing master with minimal loss of computation or state information. Chapter 8 discusses in depth the use of backup mirrors to provide fault tolerance.

5.2.3. Elements that will not transfer control

We have described various roles that the mirrors of an element can take on, and suggested that individual mirrors should be able to change their roles. Before that happens, it is important to discuss a few situations in which that may not be sensible or possible.

One circumstance where it does not make sense to attempt to transfer control of an element is in the case of elements such as TrackedElement. As discussed previously, the point of this element is to distribute the data received from a positional tracker such as those commonly used in immersive VR systems. The only difference between the master and the other mirrors is that it is on the same
site as the tracker and can therefore read data from it (via a library such as VR Juggler's Gadgeteer). Because of this, it does not make sense to change which mirror is Master – it must always be the one hosting the tracking hardware the element reads from.

More generally, this is true of all the elements that form a user's representation to Datura - usually a combination of TrackedElements, manipulators, avatars, and so forth. The user can't move from one site to another, and generally the user's representation cannot either. If the user's site is suddenly removed from the system, it probably does not make sense to attempt to transfer control of the user's avatar to another site.

How should such entities be treated? If a user's site leaves the CVE, it is probably safe to assume that user has left as well, at least for the moment. The element mirrors that made up the user's representation should shut themselves down cleanly at each site and then remove themselves from the Datura repository. For example, manipulators should release any entities they were interacting with. Avatars should remove their visual representation from the scene – whether by fading away, walking out of sight of other users, or disappearing in a Star Trek transporter special effect.

5.3. Interaction patterns for element and application design

The distribution patterns described above are fundamental to the LoC-management view of a collaborative system, but they are somewhat divorced from the application's (or application developer's) point of view. In order to put together a CVE application, a developer would ask some of the following questions:

- What objects does the environment have? How can they be interacted with? These questions can be used to define the needed set of elements to implement.

- What network communications will be needed for those elements? What optimizations can be made to the communications? These questions can help define the appropriate distribution methods for the needed elements.

- To what degree are these elements tied to particular users, sites, or resources?
These questions help determine how much freedom our LoC management capabilities have for dealing with these elements.

In order to provide answers to these questions, and explore their impact on location of computation, we present a set of broad interaction types. These are useful descriptions of the sorts of interactions that occur between elements. They can help a developer understand what sorts of elements will be needed, what kinds of communications need to take place between and within elements, and how they can be distributed. For each interaction type, we will discuss the ramifications for LoC and suggest which distribution patterns (and variants) are well-suited. Naturally, this includes a description of how different kinds of elements need to use the network and computation resources, and some of the available optimization strategies (for example, ways to replicate computations at multiple sites in order to minimize network updates). The interaction types are demonstrated with examples taken from Datura's standard library of elements.

5.3.1. User representation elements

One capability that is needed for any sort of CVE is a means to represent users. A variety of avatar technologies were mentioned in section 2.2.5. In terms of our element model, this problem can be broken down into a number of components, for example:

**Tracking elements:** Users of CVEs will typically have one or more tracked positions on their bodies, such as the head and one or both hands. These may be tracking the user's physical position, or may be controlled by desktop devices such as mouse and keyboard. Tracking data can be distributed to other sites with an interface such as the Datura `TrackedElement`. Since perfect synchronization is unattainable in real-world networks, the implementation goal is to find a viable balance between accuracy, latency, and bandwidth usage.

---

7Throughout section 5.3, the breakdowns of individual elements are meant to provide only *one possible* model for creating CVE applications. This particular model is neither definitive, nor mandatory for taking advantage of the LoC techniques described in this thesis. It is, however, meant to provide a thorough
Communication among various mirrors of such an element consists of position update messages, originating from the site hosting the user and arriving at all the other sites. One important decision is when to send such updates – at a regular period, or whenever the data changes, or whenever the data changes by a certain amount. In the sample implementation, for example, updates for `TrackedElements` were sent out at regular updates, but with the update period fluctuating based on user activity.

A second design decision is what to do with the data between received updates. This is an area where techniques such as dead reckoning [Capin+97] can prove valuable.

In terms of distribution patterns, this is an obvious example of a Master/Slave relationship. One mirror of the `TrackedElement` – the one on the site hosting the user – can gain new data about the user's position directly from the VR system. This mirror (the master) is responsible for sending updates to all the other mirrors (the slaves).

This is also an example of an element where it does not make sense to transfer roles to other sites, as discussed in section 5.2.3. If another site became the master, it would still have to get the real position data for the user from the hosting site; all that is accomplished is to increase the latency of the position data by requiring an additional step in communication. Nor would transferring the Master role give us any more reliability. If the site hosting a user goes down, there is no way to get updated position information about that user, and thus no data for the master to transmit.

**Avatar elements:** In Chapter 4, we mentioned the idea of viewable elements. One example in Datura is the `AvatarElement`, which takes location information (typically provided by `TrackedElements`) and draws a visual representation of a user.

The level of communication required for such an element depends somewhat on the complexity of the visual representation. At the simplest level, this could be implemented as a Peer/Peer presentation of these LoC techniques and how they can be used in a project. The description also follows the
distribution. Aside from sending some information (model name, colors, size, etc.) whenever a new mirror is created on a new site, no other communication is performed. The visualization is handled independently on each site, using data distributed by the TrackedElements. This might create some discrepancies in the avatar's movements on different sites, but the user's tracked positions (assumably the most important points on the user's and avatar's body) will still be in agreement.

A more complex avatar visualization might benefit from a Master/Slave distribution. This allows the mirrors of the avatar element to share more detailed information - they might transmit gestures or facial expressions of the user, or keep tighter synchronization of the avatar's movements between sites (for example, calculating how much to turn the avatar's torso when the user cranes his neck around). Alternately, the CVE might need to treat the master avatar differently (this is done in the Datura sample implementation to control whether local users' avatars are drawn).

As with TrackedElements, it probably does not make sense to be able to transfer the master role to other sites.

Other user representation elements: The above elements would allow us to create a visual representation of a user. Similarly, we might want to represent the user with other senses, for example by having an element that packages and transmits the user's voice to other sites. Such an element would follow similar distribution rules to the TrackedElements described above, though its transmissions from the master to the slaves would involve more constant streaming instead of periodic updates. Another issue is the elements that represent the user's ability to manipulate the environment, which are described in the next section.

5.3.2. Manipulators and interactions

Almost as important as user representation is the ability of users to manipulate the virtual environment in some way - for example, the ability to move the furniture around, open a door, or elements that have been created and evaluated in the Datura sample implementation.
resize a model to get a better view. There are basically two parts to this sort of interaction. First, the user has to initiate interaction — select an entity and indicate what kind of manipulation to perform. The entity, in turn, needs to decide whether to allow the manipulation, which is a question of synchronization and possibly also of permission. The second step is that the entity has to actually react to the user's manipulation — moving itself in response to the user's gestures, for example.

**Initiating manipulation (manipulators and targets):** As seen in section 4.1.2, the initiation stage of an interaction is taken care of by two kinds of elements. First is the manipulator, which represents a user's input. The second is the target, which is attached to an entity that can be manipulated. Datura's `ManipulatorTarget` is responsible for deciding whether to allow manipulation, and for keeping track of which users are currently manipulating its entity.

When a user tries to manipulate an entity, his `ManipulatorElement` makes a request to the target, which can choose to accept or reject the manipulation. If accepted, manipulation continues until the manipulator or the target decide to end it.

As with the user representation elements we examined previously, `ManipulatorElement`s are best suited for Master/Slave distribution without the ability to transfer the Master role to other sites. The target, on the other hand, is more interesting. Since it needs to make synchronization-related decisions, such as what to do when several people try to initiate manipulations at approximately the same time, it also has a M/S bent. However, it is entirely reasonable to want to be able to transfer the Master role between sites in certain circumstances. Moreover, it is interesting to consider how such a transfer could impact any manipulators trying to initiate or continue a manipulation at the same time. For these reasons, we will use Datura's `ManipulatorElement` and `ManipulatorTarget` elements extensively as an example in the following chapters.

---

8The Datura element model actually supports a third response, `ACCEPT_NONPERSISTENT`, for cases where manipulation is a single discrete event instead of a continuous change. One example is elements that can be turned on and off, such as a light switch.
Figure 5.4 demonstrates how the process of initiating a manipulation can occur when the master manipulator and master target are on different sites. Notice the use of method calls between elements on a given site, and message-passing between remote mirrors of individual elements.

**Performing manipulation (manipulation-servicing elements):** The interaction between manipulator and target elements allows manipulations to begin and end. The next step is to actually perform the manipulation. The ManipulatorTarget actually proxies for a number of other elements, each of which handles a particular type of manipulation – movement, resizing, removal, and so on. In step 5 of figure 5.4, the target element notified the (for example) GrabbableElement which actually performs manipulations. Since the target element is guaranteeing synchronization, these other elements can often be implemented as straightforward Peer/Peer distribution elements. For example, a GrabbableElement can simply follow the position data provided by the local mirror of the manipulator’s TrackedElement (figure 5.5, steps 6 and 7). The ManipulatorTarget’s synchronization will ensure that every mirror of the GrabbableElement releases the object in the
same place when manipulation ends (The messages that the manipulator and target exchange include definitive position information at the start and stop of manipulation).

More elaborate distribution might be necessary if these manipulation-performing elements are more complex – for example, if movement is tied in with a physics or collision detection simulation, which we explore below.

5.3.3. Autonomous elements

The ManipulatorTarget and manipulation-servicing elements outlined above cover cases where objects in the CVE are static – that is, they only react to user manipulations. There are also a variety of cases where an object might act on its own, to greater or lesser degree.

**Independent agents:** Some objects in a CVE application might act independently. They might be CGI-generated “robots” that perform some task, or the birds and animals in a virtual forest. Distributed entities such as these will typically have an element that controls how they move in and interact with the environment.

---

**Figure 5.5. Performing manipulation.**

6) The GrabbableElement reads the manipulator's current position from the PositionReceptor attached to it. 7) The GrabbableElement updates the Chair's PositionReceptor with new position data. Steps 6 and 7 are repeated each time the GrabbableElement updates itself, until the manipulation is terminated. Termination follows a similar pattern to steps 1-5 in figure 5.4.
A good example of this, which we will use in later chapters to demonstrate principles of controlling Location of Computation, is a `PathfinderElement` for a virtual robot. This element would plot a course through the virtual environment to some goal location, and then cause the robot entity to follow that path.

Such an autonomous element is a good example of the need for a Master/Slave distribution method, since the robot needs to follow the same apparent path on every site, and should arrive at its destination at the same time on all sites. But path planning can also be a computationally intensive task, and therefore the application might perform better if a M/PG/S distribution is used instead. By distributing path planning across multiple sites, the clock time required to find a path can be reduced\(^5\).

This is also an interesting example of where “backup sites” can be valuable. A backup mirror of the master `PathfinderElement` might store all of the master’s internal state — all the information about the goal, the complete current roadmap, how the work is being distributed to processing group members, etc. — and could take over immediately if, for example, the site hosting the master became overloaded. Backup mirrors of PG members could store partial results of computations, so that if a site was cut off from the network not all of its in-progress computations were lost.

**Semi-autonomous objects:** There are also many cases where objects should act in a semi-autonomous fashion — acting on their own, but also responding to user manipulation. The user should be able to pick up the toy robot described in the previous section. Datura’s model allows the `ManipulatorTarget` and the autonomous element to interact. For example, when the target accepts a manipulation request, it can tell the autonomous elements to become dormant so that the appropriate manipulation handler will be able to control the entity’s characteristics.

As another example, a semiautonomous element might be the element that handles a type of

\(^5\)The benefit depends on how well the path-planning algorithm can be broken down between multiple sites. The example in the test application uses separate randomized roadmaps on each site in the processing group. While each site had to create its own map, the total time to search the map was the best case of all the participants.
manipulation. Consider a physics simulation element. It could react to manipulation – allowing a user to pick up a chair, for example. It could also do collision detection on the chair, preventing it from passing through a table leg. When the user lets go, the simulation might cause the chair to tip over, fall to the ground, and so on – actions taken without direct user manipulation.

As with the fully autonomous elements described above, this could be well suited for either M/S or M/PG/S distribution. The master is useful for keeping all of the mirrors of the entity in sync: Many physics simulations are very sensitive to minor variations in the timing of input data, or to approximations in their calculations, and could drift significantly if each mirror was left to its own devices.

The PG v. slave distinction could be used in a variety of ways. The element could be written to use a processing group simply because the simulation is very computationally intensive. Another possibility is for the PG to represent those sites where physics simulations could actually be performed independently, thus providing highly accurate simulation of the object's movement on those sites, with occasional error correction courtesy of the master mirror. Meanwhile the slaves would have to rely on a simpler position approximation (perhaps using no physics or collision detection at all) with periodic corrections from the master. This might happen if some sites have much greater computational resources than others do, or if the software for performing the simulations is only available on certain platforms in a heterogeneous network.

The physics element example demonstrates how distribution choices can impact user perceptions – users on sites hosting slave mirrors of the physics element might have a much less interactive and convincing view of an object's movements than users on sites hosting PG or master mirrors. As we will see later, this can influence how we allocate those roles among the available mirrors – for example, if a user is manipulating the chair, it may make sense to make the physics mirror on that user's site a PG member, or even the master.
5.4. Distribution patterns and Location of Computation

In the context of an element model as used in Datura, controlling Location of Computation becomes primarily a question of deciding, for all the elements attached to each entity in the CVE, which mirrors will assume which roles. The interaction patterns in section 5.3 provide guidance for deciding which distribution patterns are appropriate for a given element, while the distribution patterns themselves determine which roles are available for the mirrors of an element.

In the next chapter, we describe a general location management facility, which monitors various influences that could inspire the transfer of roles from one site to another. Later chapters describe methods for actually performing that transfer of roles, both in orderly fashion and as a fault-tolerance technique, and confront the synchronization and data-integrity issues that the process can involve.
6. A Location Manager

In this chapter, we describe a general Location Manager (LM) that can exist among the sites of a CVE. It is responsible for making a variety of LoC-based decisions, and also for collecting the information necessary to make those decisions. We will explore what information to collect, policies for making LoC decisions based on that information, and methods for handling LoC-based transfers of role.

In the Datura element model, the Location Manager can itself be thought of as an element attached to a virtual “meta-object”. This allows us to think of its computations the same way we would any other element's. For example, we can specify that the Location Manager itself uses a Peer/Peer distribution pattern – that is, there is a mirror of the Location Manager at each site, and each mirror is “equal”. The Location Manager at one site has no authority over the Location Managers at any other site, but they can enter into agreements and negotiations, as we will see.

6.1. Goals of the Location Manager

In Chapter 1, we set out a group of performance and stability goals for our experiments with Location of Computation. In collecting information and attempting to perform transfers of location, the Location Manager's objectives are:

- To balance processing load among sites, in order to improve performance of individual sites (by offloading work from overworked sites) and of the system as a whole (by supporting distributed computations);
- To improve the user's perception of latency and responsiveness in the system, by performing calculations local to the user when it is beneficial to do so; and
- To provide robustness by allowing computations to be transferred to or picked up by new sites when their former hosts are removed from the CVE network.
Each mirror of the Location Manager works independently, but cooperation between them is necessary to achieve these goals. For example, if one site wishes to offload work, another will have to agree to pick it up. Section 6.3 will demonstrate how these negotiations can be handled.

6.2. Monitoring and information collection

In order to make reasonable LoC decisions, information is needed. Some information will be readily available, while other information must be collected or generated. Some will detail the local system, while some will describe the known state of other sites in the environment. This section describes several types of statistical information that can be gathered about a CVE application and the entities and elements that are part of it.

6.2.1. Local system's available resources

Every site in a CVE will have a certain amount of resources – such as computation time and memory space – that it can devote to elements. These are not simple scalar values.

Memory encompasses physical free memory, available virtual memory space, even available disk storage. But given the performance constraints of an immersive VR environment, it is seldom advisable to have data structures that cannot be resident in system memory – even a small amount of paging can be very noticeable to users.

“Computation resources” can be measured in several ways. One is to check the CPU load on a system. Another is to measure the time (either wall-clock or CPU time) that can be devoted to element computations. The latter meshes conveniently with the frame-based loops common in VR development software. For example, we could specify the available computation resources as “25 ms of total time, spread over two processors, during each 30 ms frame”.

In the sample implementation, these values can be explicitly controlled per-site, rather than determined from the state of the system. This allows the simulation of a wide variety of resource constraints for testing.
6.2.2. Local elements' required resources

Each element requires a certain amount of all available resources. The resources required by the mirror of an element at a given site depend on that mirror's role. For example, the master of an artificial intelligence element is likely to be computationally and memory intensive – much more so than the slaves. If the element uses the idea of backup mirrors, the backup is likely to have light computational requirements, but could have memory requirements approaching the master's.

Because of this, each site needs to keep resource usage statistics for the mirrors it hosts. In the sample implementation, this is collected as a short-term average (STA) of the computation required for each mirror's update during the last several seconds. The STA helps shield the Location Manager's decision making routines from brief anomalies in computation time, while still allowing it to respond to more lasting trends in computation requirements.

Memory usage of a particular element mirror is more difficult to determine automatically. The sample implementation puts the burden on the element implementation to be able to give a reasonable approximation of its current memory usage.

6.2.3. Local elements' current activities

Measuring current values and recent changes in resource usage can help the system respond to those changes after they happen, but for some elements it may be possible to know beforehand that a change is coming. A motion-planning element, for example, may know that its computation requirements will increase for a sustained length of time when it begins a new planning task. If that knowledge can be communicated to the LM, it can take proactive steps to make sure it can support the element's activity (or trade that particular role to another site).

6.2.4. Remote sites' resource statistics

Decision making will frequently require knowledge about resource usage at remote sites. For example, Site A wishes to transfer a computation to Site B. To decide whether or not to accept the transfer, site B needs to know how much of its resources would be required. The resource usage of
the mirror at Site A might answer this question, but in a heterogeneous environment this is not simple.

Computation speed can vary significantly between different computer systems, and not always in an easily-predictable manner. Memory usage can also vary, for example when moving complex data structures between 32-bit and 64-bit systems. The Datura sample implementation can apply a scaling factor to resource statistics that are shared between sites. However, this scale is currently configured manually at each site. It is not entirely clear how to automatically generate scaling factors; for now we set this question aside as a topic of possible future research.

Another aspect of monitoring remote sites is detecting sites that suffer network connection failures or that are unexpectedly removed from the CVE. In Datura, the Location Manager mirrors at different sites exchange periodic “heartbeat” messages. Chapter 8 describes how elements can deal with the possibility of an important mirror (e.g. the master mirror of an M/S element) vanishing when its hosting site becomes unavailable.

6.2.5. Local control settings

Finally, the LM is likely to have a variety of internal settings and values that govern its behavior. The following examples are taken from the Datura implementation, and apply to the behavior of the LM as described in the following sections:

- **Heavy load thresholds for resources.** These are used for load balancing [Murthy/Manimaran01]. The goal of load balancing is to distribute the workload of a distributed system across all its sites, so that resources are used efficiently and no site is overburdened. In this case, when a site's usage of some resource exceeds the threshold, the LM tries to offload some roles of the site's mirrors, transferring those roles to less burdened sites. The heavy load threshold for a particular resource is usually slightly below the maximum available.
• **Light load thresholds.** The light load threshold indicates that a site has plentiful available resources. Sites that are below their light thresholds for various resources will usually be willing to accept transfers of more resource-intensive roles.

• **Weight factors,** to control the relative importance of different factors in making location decisions.

### 6.3. Basic LM decision making

The simplest form of decision-making the Location Manager can perform is basic reactive load-balancing, based on reacting to changes in elements' resource usage. That process is described in this section; it is the first of a series of interoperating decision-making algorithms. Chapter 7 describes how roles are transferred once the LM decides to do so, while Chapter 8 expands on this by detailing how transfers can provide for error-recovery of the CVE system. Chapter 9 extends the decision-making concept in the LM to encompass user- and system-specific reasons for locating particular roles at particular sites.

The load balancing presented here is fairly naïve, and is very similar to load balancing techniques used in the wider world of distributed computing (see, for example, [Murthy/Manimaran01b], on which this particular discussion is based). However, the interactive nature of VR, with users present at all sites, as well as the heterogeneous nature of man CVEs, create additional complications for managing LoC. We will highlight particular considerations for VR systems in the discussion below. The addition of specifically user-centric considerations in Chapter 9 will transform the process.

#### 6.3.1. Load evaluation process

As described above, the LM on a site collects resource usage statistics for all element mirrors
on that site. The LM periodically compares resource usage to the heavy load thresholds\(^{10}\). If the threshold for a resource is exceeded, the LM will attempt to transfer some mirrors' roles to other sites in order to reduce resource usage.

First, the LM determines the amount of reduction necessary (the difference between the threshold and the current load for a resource). Then it selects a set of candidate elements, for which it will try to transfer roles to other sites.

The candidates could be chosen in a variety of ways. A trivial approach is simply to take the first \(n\) eligible candidates, working iteratively to sufficiently lighten the load. A more sophisticated approach could search for the smallest set that will suffice, with a goal of minimizing the message passing and negotiation of the transfers. These criteria are common in load-balancing tasks, and are among those used in the sample implementation.

Datura's element-based model includes a number of particular factors that could influence which mirrors are selected as candidates. One such factor is the role currently held by a potential candidate, and the role it would assume after the transfer. For example, a master or processor mirror could transfer that role and become a slave. A slave that's serving as a backup mirror (as described in section 5.2.2) can give up the backup and become a plain slave. However, mirrors that are already slaves aren't viable candidates, since there is no less resource-intensive role for them to assume. Similarly, mirrors of peer-to-peer elements aren't candidates, since peer is the only role available to them.

Sometimes it simply isn't possible to transfer the Master role to another site. This is the case for user representation elements such as those described in section 5.2.3 and 5.3.1.

In other cases, transferring the role of a given mirror might be possible but undesirable, simply because of the amount of data that would have to be transferred. More interestingly from a VR

\(^{10}\)As a peer-to-peer element, the LM in Datura has an update method that is periodically called by the
perspective, moving a particular role might be undesirable because it could interfere with a user's activities in the environment. Chapter 9 will describe specifically how to choose elements that should not be transferred for this reason.

6.3.2. Negotiating for location transfers

Once the set of candidates are chosen, the LM needs to find some site or sites to transfer their roles to. This can be accomplished with a simple bidding algorithm [Murthy/Manimaran01c]:

1. For each candidate, the LM sends out on the network a request for bids, essentially asking if there are any other sites in the CVE willing to take the candidate mirror's roles off its hands. The request includes information about the current resource usage of the candidate.

2. Each remote site's LM decides whether or not to send a bid. An LM will send a bid if it can accept the new role while still keeping its total resource usage below its light load threshold. If the remote LM sends a bid, it will also reserve the appropriate level of resources, so that it will not over-commit itself.

3. The original LM will receive some number of bids and pick one to accept. The simplest implementation is to accept the first bid received. A more sophisticated LM might look for a particularly well-suited bid – for example, one from a site with a high-bandwidth connection. A particularly VR-centric evaluation might look at the latency of communications between the bidding site and itself (or the sites of other users likely to interact with the candidate element). The originating LM sends a “bid accepted” message for whichever bid it chooses.

4. The LM on the accepted site prepares for the actual transfer. Other bidding sites seeing the message recognize it as a rejection, and remove any reservations they

CVE application. See section 4.1.3.
made for that bid. Bidding sites that did not see the acceptance message expire their reservations after a fixed timeout period.

5. The originating site performs the transfer to the accepting site. This process is covered in Chapter 7.

If the originating site receives no bids, it can send out a fresh request for bids after a wait period. The LM could also choose an alternate candidate or set of candidates, instead of soliciting bids for the same candidate again.

In VR applications, interactivity concerns can shape the behavior of the system in many ways. In addition to the network latencies that impact interactivity, the downtime during which a role is being transferred, and during which it cannot be interacted with, is an important user concern. In Chapters 7 and 8 we will see what factors influence this “unavailable time”, and then explore how to consider it in our LoC evaluations.

6.4. Summary

This chapter introduced the concept of a general Location Manager for an element-based CVE system such as Datura. Its data-collection capabilities were described, and a naïve algorithm for load-balancing of element roles was presented. Several of the following chapters offer significant expansions to the Location Manager's capabilities, giving it new responsibilities and adding new information sources that it can act upon.
7. Controlled transfers of location

The Location Manager described in the previous chapter can make decisions about when to transfer a particular role for a particular element from one site to another, using a variety of criteria to decide when and where a transfer occurs. This chapter explores the actual process of transferring roles – what the requirements and implications are, how transfers impact the behavior of the CVE as a whole, and what communications patterns are necessary to perform the transfers.

This chapter is concerned only with “controlled” transfers; that is: transfers that occur under the direction of the Location Managers at two sites, with one site agreeing to accept the role being given up by the other (for example, as a result of the balancing and bidding algorithm outlined in Chapter 6). We also describe how other sites react to such a transfer. There can also be “uncontrolled” transfers, which happen when the mirror acting in an important role suddenly becomes unable to do so and some other site must take over. Such reactive transfers of control are the subject of Chapter 8.

After discussing the high-level design of controlled transfers, we will briefly trace through real examples taken from the Datura sample implementation. The chapter closes with a consideration of performance issues for controlled transfers.

7.1. Goals of the transfer procedure

Ideally, the transfer of a role from one site to another should be transparent to end users, except insofar as it might be accompanied by a change in performance or latency. We can elaborate on this to produce a number of specific goals for transfers in our element-based CVE design:

7.1.1. Preservation of master state

The first, and most obvious, goal during a transfer of role is the preservation of the omnipresent entity’s state. For example, if an entity has a particular location and orientation before one of its elements transfers a role, it should still have that location and orientation after the transfer; it should not arbitrarily jump to some other coordinates.
Each element attached to an omnipresent entity will define some portion of that entity's state, as we elaborated on in Chapter 4. In the M/S and M/PG/S distribution patterns, the authoritative state for an element is provided by its master mirror. Thus, if the Master role is transferred from one site to another, all of the definitive state information owned by the master should be transferred along with it.

Sometimes, transferring every piece of state information may be impractical – consider AI or physics elements that may simply have too much state to transfer without causing a considerable interruption. As an alternative, some elements may be written only to transfer some critical set of information. For example, a path-planning element might transfer data about its goals and the paths currently being pursued, but discard the meta-data (roadmaps, space diagrams, etc.) which were used to derive paths. This data could be reconstructed by the new master if it is still needed.

In general, transferring the full master state should be the default assumption when designing elements – it will guarantee proper behavior when using the transfer technique described in this chapter. However, transferring partial state, or falling back to a safe state, may be pursued as an optimization strategy in some cases. The choice depends on a comparison of costs – the time cost of sending state data across the network, versus the computation cost of reconstructing it at the new site. These choices can be weighed by the designer when a new element is implemented, and can be compared analytically or experimentally.

The idea of falling back to a “safe” state is also valuable when performing uncontrolled transfers, and we will return to this idea in the next chapter.

Table 7.1 provides concrete examples of internal state for various elements in the Datura sample implementation and test applications, and indicates how those pieces of state information are handled during transfers.
<table>
<thead>
<tr>
<th><strong>Element</strong></th>
<th><strong>State Information</strong></th>
<th><strong>Treatment</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>ManipulationTarget</td>
<td>Current manipulation source</td>
<td>Sent during role transfer</td>
</tr>
<tr>
<td></td>
<td>Manipulator &quot;Handle&quot; position</td>
<td>Exists on all mirrors prior to transfer</td>
</tr>
<tr>
<td>LoadSimulatorElement</td>
<td>Load settings</td>
<td>Sent during role transfer</td>
</tr>
<tr>
<td>(simulates high-load computation)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PathfinderElement</td>
<td>Current &quot;step&quot; of Path</td>
<td>Exists on all mirrors prior to transfer</td>
</tr>
<tr>
<td>(finds a path in 4D space and follows it)</td>
<td>Complete current Path Roadmap Obstacle information</td>
<td>Sent during role transfer Recreated at new master Recreated at new master</td>
</tr>
<tr>
<td>PathfinderElement with</td>
<td>Current &quot;step&quot; of Path</td>
<td>Exists on all mirrors prior to transfer</td>
</tr>
<tr>
<td>backup (as above, but with a backup site for the master’s data)</td>
<td>Complete current Path Roadmap Obstacle information</td>
<td>Stored at backup site Stored at backup site Recreated at backup site</td>
</tr>
</tbody>
</table>

Table 7.1. Examples of master state treatment during transfer. In general, any particular part of the Master's internal state is either transmitted as part of the transfer or recreated on the new master. However, some pieces of information may be known to already exist on remote mirrors as a normal consequence of the element's operation.

7.1.2. Correct functioning of all other mirrors at all sites

When one mirror transfers the Master role to another, it is important that all the other mirrors continue functioning correctly. There are two main considerations:

First, all the other mirrors (slaves and processing group members alike) need to be informed of the transfer. They need to be able to identify the new master, so that they know where to send messages (e.g. requests to begin manipulations and so forth).

Second, they must not be left in any kind of "bad state". For example, imagine that a ManipulationTarget slave consults its master for permission to begin a manipulation at about the same time that the master is being transferred from one site to another. It might send the request to the old master before it learns about the new master. The ManipulationTarget element needs to deal with this in some manner. For example, the slave can timeout its pending request after a certain time (as in the sample implementation), or re-send it when it learns of the new master. In either case, the slave could recover from its momentary confusion and continue functioning correctly.

There is an interplay between this goal and the first goal (preserving master state). If the

---

11 The PathfinderElement was first introduced in section 5.3.3, and is fully described in Chapter 10.
master state were not preserved correctly, it would be easy for slaves to become out of sync with the new master. For example, the slaves might think a manipulation is ongoing, even though the new master has not maintained that information.

7.1.3. Preservation of partial computations

Another form of these master/slave consistency problems can occur with processing group members, for those elements that use the M/PG/S distribution pattern. PG member mirrors typically have to carry out all the tasks of slave mirrors, so the concerns in 7.1.2 also apply to them. Another concern, though, is what happens to a PG mirror's processing results when the Master role is transferred.

As with slaves, the PG mirrors need to learn which site is the new master, so that they can send their results to the right site. The new master also needs to know the locations of all the PG members, so that it can send new processing requests to them. This information should be part of the master state that is sent to the new master as part of the transfer process.

In M/S elements, only the master role can be transferred from one site to another. But in M/PG/S sites, a PG member role can also be transferred to another site. In some ways this is similar to transferring the Master role, but the dangers if all of the PG mirror's state is not transferred are usually less. Generally, this would only cause a performance hit as partial computations are lost and have to be reperformed elsewhere. This is unlikely to cause an inconsistency within the element's state.

7.1.4. Minimal disruption for users

With the more technical goals explained, we should consider again how the user fits into the scheme. Minimizing disruptions that are perceptible by the users of the environment restricts, for example, any shortcuts the system might take in maintaining master state of elements. For example, if a user is manipulating some object in the CVE, it would be undesirable for that manipulation to terminate unexpectedly because of some role transfer happening behind the scenes.
Similarly, the user could be confused and inconvenienced if the system's approach to handling a manipulation request that happens during a role transfer is to reject it out of hand. The user isn't likely to be aware of the shuffling of roles between sites in the CVE, and so it will appear that the transfer failed for no reason at all. Better for a more robust solution to be implemented behind the scenes, that will maintain the user's illusion of perfect interaction with the environment.

7.2. A strategy for performing role transfers

Given the requirements above, we present the following generic strategy for transferring a role between mirrors. The description only covers the case of controlled transfers; i.e. those where both the source and destination of the transfer have agreed to it in advance. The unique problems posed by uncontrolled transfers are addressed in the next chapter.

In general, the discussion focuses on the transfer of the Master role for M/S and M/PG/S distribution patterns. Transfer of a Processing Group role is similar in most respects, and differences are detailed where they occur.

7.2.1. Transferring the role

A controlled transfer begins when the Location Manager at a source site, and the LM at a target site, agree to transfer the role for a particular element from the source to the target site. The transfer ends when the mirror of that element at the target site has adopted the transferred role, and every other mirror (including the source) has been notified of and adjusted to the shuffling of roles.

The basic transfer occurs in four steps:

1. The source mirror changes its role from Master (or Processing Group Member) to Slave Without Master. This role is similar to the Slave role described previously, except that the mirror believes that no master currently exists. It will remain in the Slave Without Master role until the transfer is completed. The distinction of
the Slave Without Master role is important in case the transfer fails in some way; its full use is a key point of Chapter 8\textsuperscript{12}.

2. The source packages as much of its state as will be necessary for the new master to take over, given the constraints discussed above. This is then sent to the target site as a transfer message.

3. When the message is received, the mirror at the target changes its role to Master (or Processing Group Member, if that is the role being transferred) and updates its internal state with all the information contained in the transfer message. If the element is also one that recreates some of its internal state during a transfer, it does so (or at least initiates the recreation).

4. Once the target has finished adjusting its internal state representation, it needs to announce itself. If the Master role was transferred, it sends a "New Master" announcement to all mirrors. If the transferred role was instead PG member, a similar announcement is sent – but only to the current master. The reactions of the slave and PG mirrors when they receive such an announcement is detailed in sections 7.2.2 and 7.2.3.

Once step 4 is completed, the new master is "ready for business" - it can carry out all the usual tasks of a master: performing calculations or parceling them out to other sites, synchronizing data access, and so on. If the element uses the M/PG/S distribution pattern, a small amount of additional communication is undertaken to sort out the master's relationship with the Processing Group; this is seen in section 7.2.3.

\textsuperscript{12}There is, analogously, a "Processing Group Member Without Master" role, which serves the same distinction as Slave Without Master. However, note that the source during a controlled role transfer will always reduce itself to Slave Without Master, regardless of the original role.
7.2.2. Maintaining consistency of other mirrors

When the Master role is transferred from one mirror to another, every other mirror is affected indirectly. In this section, we explore the effects of such a transfer on the slave mirrors.

Notification of new master. When Slave mirrors receive notification of a new master, they make a note of the master's address so that they can direct communications to it. Slave-without-Master mirrors do the same, and also upgrade their role to Slave. In most cases, they can then continue on as if the transfer never happened.

Maintaining consistency with interrupted communications. If a given slave was attempting to interact with the master at approximately the same time as a transfer occurred, it can be more difficult to maintain consistent behavior. Consider the ManipulationTarget element described in section 5.3.2, where slaves contact the master for permission to begin a user interaction. The slave could send to what it believes is the master mirror, only to have the Master role sent somewhere else before the message is received. The former master is now in a slave role, and will reject the message instead of granting (or not granting) permission. If the former master's site was offloading roles so that it could leave the CVE, the message might never be received by anybody.

This has important consequences for any element whose slaves communicate with the master. First, any message might be "rejected" by the site that receives it. Slaves can deal with this in several ways. If the slave has received a "New Master" notification in the interim, it can simply resend the message to the new master – the result for the user is that beginning the interaction has a higher than usual latency. Alternately, it can simply accept the rejection as a denial of permission to begin the interaction. In the sample implementation, if the user tries to grab an object and a rejection happens, the grab attempt fails. If the user tries again a moment later (after the transfer has worked itself out and all the slaves know about the new master) it will succeed.\(^\text{13}\)

\(^{13}\)Either way of dealing with this disruption slightly inconveniences the user, if a role transfer is
Instead of being rejected, the request from the slave might simply “disappear” and never result in any response. To deal with this possibility, the slave must be able to “time out” messages after a period of waiting. If a new master notification happens while the slave is waiting for a response, it can skip the rest of its waiting period and immediately resend the request to the newly announced master.

If the slave chooses to respond by resending a request message, it is important not to become stuck in a loop. At some point, the slave must realize that either there is no master anywhere in the network, or that it has completely lost track of what site has the Master role. At this point, the slave would enter into the process of negotiating a new master, which we describe in the next chapter.

**Notification of regenerated or recreated state.** If the element undergoing a transfer recreates some of its internal state when a new master is created (instead of transferring all of it), the proper slave state after the transfer may not be the same as the state before the transfer. For example, a path-planning element may cancel all previous paths when a new master is created, or a manipulation element might cancel the outstanding interactions. In these cases, fresh state information could be attached to the “New Master” announcement and adopted by every other mirror that receives it.

This sort of discontinuity during a transfer may be necessary for some elements for practical reasons, for example if very large amounts of data would have to be sent to the new master otherwise, or if different implementations of an element exist on different sites. However, it should be avoided if possible when designing elements, since it is more liable to create user-noticeable changes.

Discontinuities like this are also more common when considering uncontrolled transfers, especially of elements that don't use backup mirrors.

---

occurring at the same time as a user interaction. One of the enhancements to Location Manager behavior discussed in chapter 10 will consider actively avoiding such a coincidence. Load balancing can be modified to favor transferring roles in objects the user is less likely to interact with.
7.2.3. Considerations for Processing Group maintenance

Processing group members share many of the same issues as slave mirrors described in the previous section, but there are also several unique concerns that must be addressed.

**Maintaining the Processing Group member list.** The first problem is maintenance of the processing group as a whole: ensuring that it has a sufficient number of members to handle the element's processing needs, and making sure that the (new) master mirror knows about all the available members. The initial membership of the processing group is managed using a bidding process similar to that used for controlled transfers; the master sends a request for sites that can handle the extra processing load, and tells a certain number of them to promote themselves from the Slave role to the Processing Group role. The master is responsible for deciding how many mirrors it wants in its processing group, and for deciding how often to look for new potential members. The master keeps track of all the sites that have joined the processing group.

When the Master role undergoes a controlled transfer, it is practical to send a list of PG members as part of the transfer message. If the role is being transferred to a backup mirror, then this list might only contain changes since the backup was last updated.

When one of the PG member mirrors transfers its role to another site, the master needs to be informed so that it can update its internal list. In the Datura implementation, the recipient of the transfer is responsible for announcing itself to the master.

**Maintaining the transferred role's internal state.** The second problem when handling role transfers involving PG members is maintenance of work assignments and partially completed computations. If a PG member is in the midst of a computation when a transfer is initiated, it could send some data along with the transfer message. This could range from a simple set of instructions to perform a given computation, to the full internal state of the original PG member. As with the Master role, a PG mirror can maintain a backup site to store partial results and pickup with minimal disruption when a transfer occurs.
Dealing with interrupted communications. A state-management problem occurs when the master tries to send a message to a PG member while the PG member is transferring its role to another site. This is not dissimilar to the problem with slaves sending messages to a transferring master (section 7.2.2.), though the solution is slightly different.

When a slave sends a message to a master, it's typically requesting information or permission, and expects a quick response. When a master sends a message to a PG member, it may take longer for a computation to be performed and a response to be sent. Therefore, in order to detect a communication lost during a PG member transfer, it is more important for the master to keep track of announcements sent following a transfer, and for it to resend messages if necessary. It can be helpful for PG members to send back confirmation messages immediately, so that the master knows its processing orders are being carried out.

A more difficult scenario arises as a result of a “lost” message from the master. This happens if a processing task is ongoing, and the master sends an updated request (for example, changing the destination of a path planner), and the updated request is lost while a PG member transfers its role to another site. This leads to a race condition: the new PG member might complete the (original) computation and send a result back to the Master, before the master notices the transfer and resends its updated instructions to the new PG member. The master needs to realize that the result is for the original, outdated version of the computation task and respond to it appropriately. One response to this, used in the Datura test applications, is to add a unique identification to all computation requests sent by the master, and to include that identification in all computation results sent to the master from PG member sites.

7.3. Concrete examples

Having discussed the concepts behind controlled transfers, this is a good location to pause and consider some concrete examples. In this section, we explore (in pseudocode) the role various classes in the Datura sample implementation play in managing controlled transfers.
7.3.1. Initiating a controlled transfer

In Chapter 6 we described the Location Manager and basic load balancer. The LM is responsible for initiating a controlled transfer. Once it has decided which element's role to transfer, and received an acceptable bid from a remote site, it calls a method called sendPerformRoleTransfer():

```c
void LocationManager::sendPerformRoleTransfer (Element *e, Address remote_site)
{
    Buffer data;
    // LM's identifier - used for message delivery
    writeGUID (data, mTypelD);
    // write the command
    writeUint8 (data, PERFORM_BEHAVIOR_TRANSFER);
    // write address of the new recipient
    writeAddress (data, remote_site);
    // The entity and element whose role is transferred:
    writeGUID (data, e->getBehavioredObject()->getID());
    writeGUID (data, e->getTypeID());
    // instruct the local mirror of the element to start the
    // transfer
    e->beginTransferRole (data);
    sendMessage (data);
}
```

The interesting part is the `beginTransferRole()` method of the `Element` class, which serves two purposes. First, it instructs the local mirror that it is being transferred, and that it needs to drop its current role. Second, it allows the element to write whatever data it wishes to send to its replacement into the data buffer.

To give an example of what happens in this method, here is a representation of the `beginTransferRole()` implementation in Datura's `BasicMSElement`. `BasicMSElement` is an abstract base class for elements using the M/S distribution pattern:
virtual void BasicMSElement::beginTransferRole (Buffer data) 
{
    // write our role, which is to be used at the destination site
    writeUint32 (data, mRole);

    // reduce our succession priority, so that we don't conflict
    // with the destination site after the transfer. Succession
    // priorities are used for error handling, and are described
    // in Chapter 8.
    SuccessionPriority new_priority =
        mSuccessionPriority.generateChild();
    // write our succession priority, which will be adopted by
    // the destination site.
    writePriority (data, mSuccessionPriority);
    mSuccessionPriority = new_priority;

    // change our own role. Before the transfer, our role is
    // MASTER. We drop to the SLAVE_WITHOUT_MASTER role
    // until the transfer is completed.
    assumeRole (SLAVE_WITHOUT_MASTER);
}

Obviously, this class writes only the bare minimum. Classes descending from
BasicMSElement can add more information, but for many elements this version is sufficient. For
example, the ManipulatorTarget element sends just this information. While it is important for its
new master to have the same internal state - the same ongoing manipulations, for example - this
information is already available at all its other mirrors so that they can react to user manipulation
correctly.

Other elements need more information to perform their tasks correctly after a transfer. For
example, one of the Datura test applications includes a robot called a "gravbot" which flies around in
wild patterns and interacts with nearby users (see Section 10.2 for details). It overrides
beginTransferRole to include its most current position, velocity, and forces interacting with it.
While some of this information is available to other mirrors, putting up-to-date, canonical information
in the transfer message assures a smoother transition.

Another virtual robot in the test application uses the PathfinderElement, which we've
mentioned previously. The pathfinder searches for an unobstructed path in the environment and then follows it. The PathfinderElement's beginTransferRole looks like this:

```cpp
virtual void MSPathfinderElement::beginTransferRole (Buffer data)
{
    BasicMSBehavior::beginTransferRole (data);

    // Write current state of the pathfinder AI
    writeUint32 (data, mAIState);

    // If we have a current path, send it in the transfer
    if (mAIState == FOLLOWING_PATH_TO_CARGO ||
        mAIState == FOLLOWING_PATH_TO_LANDING_PAD)
    {
        writePath (data, mCurrentPath);
        writeUint32 (data, mCurrentPathStep);
    }
}
```

### 7.3.2. Completing a transfer

The other side of beginTransferRole() is completeTransferRole(), a method of the BasicMSElement class that is called when a site receives a transfer message.

```cpp
virtual void BasicMSElement::completeTransferRole (Buffer data)
{
    int new_role = readUint32 (data);
    mSuccessionPriority = readPriority (data);
    assumeRole (new_role);
    if (new_role == MASTER)
    {
        sendNewMaster ();
    }
}
```

Again, the basic version is simple. Elements that override beginTransferRole() will do the same with completeTransferRole(), so that they read in all the extra data their mirror added to the message.

If the role being assumed is Master (as it always is for this particular class), the element next
announces itself (the sendNewMaster() call above). This allows other mirrors to know that it is ready, and they can change their roles from (for example) Slave Without Master to Slave.

If an element extends beginTransferRole() to include additional information, it must also extend completeTransferRole() to read and handle that data. For example, this is the implementation for the pathfinder we described above:

```cpp
virtual void MSPathfinder::completeTransferRole (Buffer data)
{
    BasicMSElement::completeTransferRole (data);

    // Read current state of the AI
    mAIState = readUint32 (data);

    // See if our message includes a path to follow
    if (mAIState == FOLLOWING_PATH_TO_CARGO ||
        mAIState == FOLLOWING_PATH_TO_LANDING_PAD)
    {
        mCurrentPath = readPath (data);
        mCurrentPathStep = readUint32 (data);
    }
}
```

Controlled transfers are by far the simplest kind. Uncontrolled transfers, which we will see in Chapter 8, require more elaborate considerations.

### 7.4. Performance effects of controlled transfers

Controlled transfers are most often performed to improve the performance and/or interactivity of the system in some way. Yet the act of transferring a role from one site to another also has its own costs to consider, and moving a role may have other indirect effects on performance. Additionally, the way that the roles in a set of elements, as a whole, are distributed amongst the available sites can impact the load on individual sites in interesting ways. Some of these factors can also be measured by the Location Managers at each site, and used to influence how location choices are made.

#### 7.4.1. Transfer design and unavailable time

The most important thing to remember about transfers, especially transfers of the Master role,
is that while the transfer is underway, the master is unavailable. The total time of unavailability is
determined by the size of the transfer message, along with the bandwidth and latency of the
connection between the sender and receiver. However, remember that the new master needs to
announce itself to the other mirrors. For any slave mirror, the perceived unavailable time due to a
transfer is increased by the time required for the new master announcement to reach it. Thus, the
unavailable time may appear slightly different at different sites on the network.

7.4.2. Effects on user-detectable interaction latency

A transfer of role can cause changes in the responsiveness of the CVE as experience by a user,
primarily by changing the latency related to operations. Remember, initiating some types of
interaction, such as grabbing an object, requires getting permission from the master
ManipulationTarget attached to that entity. If that particular master is being transferred at the
same time, it won't be able to give (or refuse) permission until the transfer completes. Moreover, the
mirror requesting interaction may have to resend its request after it receives notification of the new
master. In these uncommon circumstances, the latency associated with beginning an interaction is
increased by up to the entire unavailable time, plus the time to resend the request.

There is a smaller latency impact even without the sort of collisions described above. Moving
the master role from one site to another can change the time required to send and receive messages
from other sites, changing the time required for a particular site to request permission. In the best
case, of course, the master will be on the same site as the requester.

The enhanced Location Manager techniques in Chapter 9 will consider how to take advantage
of these variations, attempting to actively reduce latency noticeable to users of the CVE.

7.4.3. The big picture: Distribution of computation and responsibility

Of course, the entire impetus for being able to control location of computation is to improve
performance of the CVE as a whole. Even the basic load-balancing presented in the previous chapter
delivers certain advantages. To understand this, we can look at how roles are spread throughout the
system as a result of controlled transfers driven by load-balancing.

Figure 7.1 presents a sample CVE network, showing where the master roles for certain elements are located initially, and then showing one possible way they could be distributed as the system runs.

In the second diagram, the Master roles have been spread around among all the available sites. This gives several major advantages to the performance and stability of the system:

- The direct effect of the basic load balancing algorithm is that the processing load associated with hosting Master and PG Member roles is spread out across the sites of the system.

- An important side effect of this distribution is that the flow of network traffic is also spread out across the network. When one site hosts all or most of the master mirrors in the system, all requests from slaves to masters must be sent to that particular site. After distribution, requests involving different elements are sent to different sites, reducing the network pressure on site A.

- Responsibilities are no longer centrally located at site A. If no transfers happened, and site A vanished, almost every element in the CVE would have to go into an error-recovery mode (the details of which are explored in Chapter 8). The controlled transfers reduce the cost of such a failure. Fewer elements have to perform uncontrolled transfers, which means less network traffic as they sort themselves out. Also, fewer elements will be in unavailable time while those transfers happen, meaning that there is less likely to be a noticeable effect on any particular user. Finally, this reduces the risk of losing important state information in elements, limiting the inconsistencies that could arise as the result of losing the vanished masters.
Figure 7.1. Distribution of master mirrors of elements. The top picture represents the initial state of a CVE; the bottom represents the changes in location of various roles as a result of controlled transfers. Only master mirrors are shown in this picture; slaves are omitted.
7.4.4. Location Manager metrics for packaging and transferring state

Transfers of role and their associated computations create additional metrics that can be considered by the Location Managers on each site participating in the collaboration. We list those metrics here; Chapter 9 considers how our location-control algorithm can take them into consideration.

Bandwidth requirements for transfers. Elements can be made to provide estimates for the amount of data that would have to be transferred when (for example) the Master role is transferred from one site to another. For some elements, this will be a fixed amount. For others, it may vary based on the current activities being undertaken, or algorithms being computed. For elements that are transferring a role to a backup mirror, the amount of data to transfer may also depend on the time since the backup's data was most recently updated.

Approximate time for a transfer. When considering whether to perform a transfer, or evaluating bids as described in the previous chapter, the Location Manager can also approximate the time that would be required to perform a transfer – part of the unavailable time we mentioned above. This is actually a combination of several factors: The quantity of data for transfer, described above; and the bandwidth and latency of network connection from the transferring site to a potential target. Whether the role being transferred is Master or Processing Group Member, this transfer time also indicates how long a computation will be disrupted, or how long the user might be unable to interact with an object. An additional aspect of this is how long it will take for news of the transfer to propagate to the other sites in the network (mostly a question of the network connectivity and latency).

7.5. Checkpoint

The information in this chapter is sufficient for designing elements of omnipresent entities that can successfully transfer roles from one site to another under controlled circumstances. There are a variety of ways that a CVE can enter an “uncontrolled” state, which could cause transfers to be
interrupted or force them to happen in a more chaotic environment. These situations, and enhancements to maintain consistency and functionality of elements when they occur, are detailed in the next chapter.
8. Uncontrolled transfers of location

Roles are not always transferred under ideal, or controlled, circumstances. Sometimes the mirror holding a particular role vanishes unexpectedly, leaving the other mirrors to pick up the pieces. This chapter describes how and why such disruptions happen, and provides a mechanism for restoring a needed role at a new location. The key concepts are creating a consensus among the remaining mirrors about what to do, and restoring a consistent state to the element as a whole.

We begin by discussing why uncontrolled transfers happen and how they can be handled. This description is followed by a short look at real examples from the Datura implementation. The chapter closes with a consideration of the performance concerns related to uncontrolled transfers.

8.1. Causes of uncontrolled transfers

A variety of failures can force a CVE to engage in uncontrolled transfers. A network connection might fail or time out, a computer could crash, a plug could be pulled, etc. The result is that one or more sites are removed from the CVE without providing any notification to the remaining sites. Alternately, the CVE might be split into two separate, but unconnected, sets of sites. We refer to sites that leave the CVE without any announcement or warning as "vanished" sites. Central to this idea is that entire sites vanish, with all their mirrors.

The severity of such a disruption can be considered separately for each element of each omnipresent entity in the CVE. The severity for any particular element depends on its distribution pattern, and on which roles were hosted by the missing sites.

Mirrors using the Slave and Peer roles cause relatively little disruption – there is no need to have some other site assume the role of the vanished mirror. There is generally no concern about consistency when slaves mirrors vanish. Hypothetically, a single vanishing mirror might cause

\[\text{In the case of such splits, we consider each group of sites separately – the result is just like a large group of sites leaving the CVE at once. Each group will consider all the sites in the other group as vanished. A}\]
consistency problems for other elements (consider the communication pattern we described in Figure 5.4) — but only on the same site as the vanished slave. Our model disregards this possibility, as only entire sites can vanish.

Processing group members are a slightly more complicated problem. Obviously, a vanished PG member will never send a result for any computations assigned to it by its master. The master needs to detect vanished PG members so that the computations can be reassigned to some other site, and so that the master does not wait forever for a result that never arrives.

Vanished master mirrors are the most serious problem a CVE can face. The master contains definitive state for an element, and is responsible for decision making. If the master vanishes, it's up to the remaining mirrors to appoint a new master, and restore the element's internal state as much as possible. This could also involve negotiation with other elements that were being interacted with (such as the ManipulatorElement of one entity interacting with the ManipulatorTarget element of another) so that the CVE as a whole is kept consistent. The act of appointing a new master is called succession negotiation, and is the topic of section 8.4. Section 8.5 describes dealing with consistency problems.

Finally, another problem condition can occur if the Master role is being (controlled) transferred and the destination site vanishes. The element now has no master, but this situation is somewhat more difficult to detect, as we'll see in the next section.

8.2. Detecting vanishing sites

The first step in dealing with a vanishing site is detecting that the site has vanished. There are a number of ways this could happen:

8.2.1. Location Manager heartbeat

One method that allows vanished sites to be detected within a known length of time is to have
each site broadcast a periodic heartbeat signal—a short message indicating that the given site is still alive. The Location Managers at each site can listen for these heartbeat signals; if a sufficient interval passes without a new heartbeat from a particular other site, the LM can notify all the mirrors at its site that the other site has vanished. The individual mirrors generally know who their master site is, and can therefore decide how to respond to the vanishing site.

This method of detecting vanished sites is not always sufficient, because there are times when a mirror won't know who its master site is, and therefore won't be able to respond correctly. For example, this could happen if the Master role was being transferred to the site that vanished—if the transfer never completes, the other mirrors (slaves and PG members) will never learn that the vanished site was supposed to be the master, and won't respond as they should. Fortunately, this situation can only arise through a very specific timing of events, and is not likely to occur often.

8.2.2. Noticing unanswered messages

Another way to detect that an important site has vanished is to notice messages that don't receive a response. Simply, a slave mirror that sends a message to its master can wait a fixed time for a response. This provides a backup strategy in the event of a timing problem like that described above. In this case, the slave won't detect the problem until it tries to interact with its master in some way. This reactive detection is less desirable, since it is more likely to cause a user-noticeable interaction problem, and thus should not be the primary means of detecting vanished sites.

8.2.3. Detecting networking failures

Depending on the network infrastructure and software underlying the CVE, it may be possible for some sites to directly detect when connections to other sites fail (e.g., by receiving an exception or failure code from the networking software). The Location Manager at the detecting site could then forward information about the vanished site to the rest of the sites in the CVE.

The usefulness of this detection method depends on the features of the underlying network library, and the conditions that cause a particular site to vanish. Because of this, it should not be
relied on as the sole means of detecting vanished sites. On the other hand, it may offer more immediate detection than the techniques mentioned above.

8.2.4. Verifying vanished sites

To avoid confusion amongst the mirrors of a CVE, it is important to verify that a vanishing site is actually gone. False alarms might happen because of a temporary communications failure or bottleneck, especially since several of the methods described for detecting missing sites are timing-based. At the same time, it is important that the CVE respond quickly to vanished sites so as to not inconvenience users. In the succession negotiation methods we'll explore below, it is always possible for the original "vanished" master to reassert itself and short circuit the negotiation process.

8.3. Handling vanished PG members

Handling vanished PG members is a relatively simple affair that can be entirely dealt with by the master site. Any of the methods in section 8.2 can be used to detect a vanished site hosting a PG member. The master then checks the vanished site's name versus the list of sites it knows to be hosting PG members. If there is a match, the master knows that one of its PG members has (apparently) vanished, and can take steps to create a replacement.

8.3.1. Promoting a new Processing Group member

We previously described how the master in a M/PG/S distribution pattern would periodically request new PG members from its slave mirrors. The process for dealing with a vanished site is similar, except that the request the master sends out includes one additional piece of information: the name of the vanished site.

This solicitation from the master can garner a variety of responses from the other mirrors:

---

15Any network infrastructure will provide a way of uniquely naming a site. The combination of IP address and a port number is an obvious example, which allows multiple "sites" to be hosted on a single computer.
• If the vanished site was a false alarm, the “vanished” PG member can speak up, assuring the master that it still exists. This reply should also include checkpoint identification (as mentioned in section 7.2.3). This situation can also arise if the PG role was undergoing a controlled transfer at the same time the master sent a message to it, and the master's message was lost – the site reasserting itself is the destination of the controlled transfer.

• If the vanished site had a backup mirror, the backup can identify itself to the master, again including a checkpoint ID to indicate when it was last updated by the vanished PG member.

• Slave mirrors on lightly-loaded sites can send bids, indicating that they can take on the extra processing load of being promoted to PG members.

• Slaves on heavily-loaded sites will not send any response.

The master waits a fixed time for responses, though it may short-circuit this if it receives a response from the “vanished” PG member or its backup. Ultimately, the master selects a new PG member (preferring the original, if it is found to still exist; followed by its backup, if any; and then selecting from slaves that sent bids). It sends the new member a message announcing its choice, and indicating any processing that it should begin performing. The particular contents of this message may depend on the checkpoint that the site sent to the master. For example, if the master promotes a backup mirror, it may simply tell it to proceed with computations based on the state the backup had inherited from the vanished PG member. On the other hand, if the checkpoint indicates a computation that is no longer relevant, or if the master is promoting a slave, it may send information detailing an entirely new computation task.

As for the other sites, such as slaves that were not selected for promotion, they can ultimately let their bids expire. If the underlying network infrastructure allows multicasting or broadcasting at
low cost (as in the sample implementation), they may see the master's message promoting another site, and immediately remove their (rejected) bids.

8.3.2. Managing incorrect promotions

The strategy outlined above allows processing groups to be rebuilt very quickly in the event of a failure. However, it is possible to imagine scenarios where it does the wrong thing. For example, if a network disruption occurs for an extended period of time, cutting off a PG member site from the rest of the sites in the CVE, the master might carry out the entire promotion process described above before the PG member regains communication. The result is that the temporarily vanished PG member might think it is still a PG member, while the master has given up on it.

Fortunately, the consequences of this inconsistency are relatively minor. On the down side, the temporarily vanished PG member might continue with a computation that the master has reassigned to some other site. Also, the PG member won't respond for requests for new PG members (since it thinks it already is one), meaning that its processing resources will basically become unavailable for that particular element.

This situation can be detected under certain circumstances. For example, if the vanished site was engaged in a computation, it will eventually send results to the master site – which can then decide what to do with it. If the site was cut off from the rest of the CVE for long enough, it may also have engaged in succession negotiations to find a new master for its own subset of the CVE (as described below). When communication is restored between the severed sites, the element as a whole might have two masters on separate sites. Section 8.4.4 describes resolution strategies for these sorts of conflicts.

8.4. Handling vanished masters

A vanished master mirror is the most difficult situation to account for. When PG members vanish, the master can act as a central authority to collect bids and promote a successor to the vanished mirror. When the master vanishes, there is no such central authority. All that remains is a
set of slave (and possibly PG member) mirrors – a set that may not even be of known size. Somehow, these mirrors must promote one of their number to the Master role and restore the element’s state as closely as possible. The entire process of detecting a missing master, through appointing a new one and restoring internal consistency, is referred to as succession negotiation.

### 8.4.1. Beginning succession negotiations

Succession negotiation is a process that takes place across the CVE, but it is also a condition than an individual mirror finds itself in. A slave mirror or processing group member is involved in succession negotiations starting when it notices its master mirror has vanished, and ending when it receives an announcement of a newly-chosen master (or it adopts the Master role itself).

When a mirror detects that its master is missing, using any of the techniques listed in section 8.2, it changes its role from Slave or PG Member to “Slave without Master” or “PG Member without Master”. It then broadcasts a message to its other mirrors indicating that it thinks the master is missing. Mirrors receiving this message change their own roles to the “without Master” versions, and succession negotiation begins.

If, for some reason, the missing master indication was a false alarm, the master will receive the message along with all the other mirrors and immediately respond, quelling all uncertainty among the mirrors.

### 8.4.2. Succession priorities

Early in the Datura design process, voting algorithms [Pfleeger97] were considered as a means of appointing a new master mirror. Unfortunately, several problems presented themselves with this method, mostly due to the fact that voting systems need to know the exact size of the set of “voters” in order to reach a conclusion. The situations that lead to uncontrolled transfers are likely to change the number of available mirrors in unpredictable ways.

Since voting strategies for succession management ultimately appeared impractical, an alternative method was found using succession priorities. Each mirror of an element is given a
unique priority, stored as a sequence of integers. The current master at any given time is responsible for assigning priorities to any mirror that requests them. The primordial master is given the priority ‘0’; it hands out priority numbers such as ‘0:1’, ‘0:12’, and so on. Any other site, should it become the Master, prepends its own priority to the numbers it assigns to other mirrors (e.g., a mirror with priority ‘0:12’ would hand out numbers such as ‘0:12:3’ or ‘0:12:6’). Priority numbers are fully ordered; for example: ‘0’ > ‘0:1’ > ‘0:4’ > ‘0:1:3’ > ‘0:4:3’ (note that a shorter sequence is always higher priority than a longer sequence).

The reason for the multipart priority number is to ensure uniqueness in the assigned numbers even under extreme circumstances. For example, imagine that a master hands out a priority number to some new mirror and then vanishes. Another mirror will become the master after succession negotiations, and that new master will be responsible for handing out priority numbers whenever a new site joins the CVE. The new master may not be aware of all the numbers handed out by its predecessor, but by using its own priority number as the base for the new ones it hands out it guarantees that no collisions can occur.

Using these priority numbers in succession negotiations is simple: the highest-priority mirror can take control and become the new master. As a rule, the highest priority mirror is always the master. Lower priority mirrors will always accept a higher-priority mirror as master. We outline the process of using these numbers in the following sections.

8.4.3 Finding the highest-priority mirror

The highest-priority mirror must announce itself to the other mirrors. Unfortunately, a mirror is seldom in the position of being able to declare absolutely that it has the highest priority. To prevent every mirror from announcing itself at once and flooding the network, each mirror calculates a wait period before it will send an announcement declaring itself the new master. A simple heuristic is used to pick a wait value such that low priority mirrors are unlikely to send before high priority
mirrors. It is calculated by checking the length of the mirror's priority number sequence versus that of
the last known master, and estimating the amount of branching at each level of the sequence.\textsuperscript{16}

Once a particular mirror's waiting period is expired, it sends a message to the other mirrors
announcing itself as the new Master.

\textbf{8.4.4. Handling conflicts}

The heuristic delay system described above can be subverted in a variety of ways. A particular
mirror might estimate its position poorly, or slow networks could simply cause a high priority mirror's
announcement message to arrive after another mirror has declared itself master. Conflicts are
resolved by comparing priorities; a mirror that receives conflicting new master announcements
honors the one with the highest priority. A mirror that receives a new master announcement with a
priority lower than itself immediately takes over, sending out its own new master message to override
the "usurper".

The overall result of this is that, in rare cases, the Master role might fly between a number of
mirrors before settling in one place; it is also possible for two sites to be Master at the same time (for
a very brief period).

Another conflict resolution problem happens with network splits. If a site that connects two
other sites, or two groups of sites, vanishes, the CVE is split into two pieces. Each will adopt its own
masters using the method described above. If the two groups are rejoined later, the two masters for a
given element will eventually learn about each other. The one with the highest priority will send an
announcement overriding the usurper. This event can cause noticeable inconsistencies for users on
the same side of the split as the low-priority master; see section 8.5 for a discussion.

\textbf{8.4.5. Declining promotion to Master}

A mirror on a heavily-loaded site may not be able to assume the Master role even if it does

\textsuperscript{16}This is almost like treating the space of available priority numbers as a tree with variable branching,
have the highest priority. This can be dealt with in a variety of ways. The simplest implementation is to force mirrors to take the responsibility regardless of load. Once the element has stabilized, normal load balancing can cause a controlled transfer to a less heavily loaded site. This is the method used in the sample implementation.

An alternate method is to simply stay silent and wait for a lower priority mirror to declare itself Master. Once it does so, instead of overriding the usurper as described above, the mirror that declined the Master role can abandon its original priority and request a new priority number from the new Master. This preserves the constraint that the master has the highest priority in the system.

8.4.6. Managing priorities during controlled transfers

A careful reading of the priority constraints suggests that, any time a controlled transfer occurs, the original master should consider itself usurped and immediately try to take control again. To prevent this, the old master sends its priority number along with the other data in the transfer message. The new master adopts the old master's priority, and starts using that priority number as the prefix for any priority numbers it hands out to newly-arrived mirrors. This behavior was demonstrated in the code examples in Section 7.3.

The old master also needs to assume a new, lower, priority. In the sample implementation, it creates a new priority for itself the same way it would for a newly created mirror. Of course, this must be done before the priority is written into the transfer message.

8.4.7. Priorities and backup masters

Handling succession negotiations for elements that use backup masters requires some special consideration. We must allow the backup to assume control when the master vanishes, but our and estimating a particular node's order in a traversal.

\textsuperscript{17}This assumes that there is a lower-priority mirror on some site. If not, the high-priority mirror can always wait and declare itself master if the load at its site lightens enough to allow it – or even ask its local Location Manager to lighten the load so that it can take over the Master role.
default priority assignment method doesn't guarantee that the backup will be next in line of succession.

The solution we use is to add extra information to the priority number. A master with a priority of, say, '0:4', can grant its backup a new priority of '0:4a', which is considered a higher priority than (for example) '0:5' or '0:4:1', and lower than '0:4'. With this priority, our succession negotiation can proceed normally. The new master, with priority '0:4a', can then choose its own backup mirror and give it the priority '0:4b', and so on.

An important point to make is that, since we still use succession priorities, elements using backup masters can still recover correctly if both the master and its backup vanish.

8.4.8. Other uses of priority numbers

While the use of unique priority numbers requires a certain amount of setup by the elements that use them, there are also interesting ways that these numbers could be taken advantage of. For example, by being responsible for handing out priority numbers, the master mirror can control which mirrors can take part in succession negotiations. In the future, this could be a building block for an authorization mechanism, with the master handing out succession priorities only to those mirrors it trusts to take over its role.

8.5. Maintaining consistency and resolving state conflicts

Uncontrolled transfers are essentially a form of error recovery. A well-designed element will pick up the pieces of its internal state as well as possible and then carry on. Ideally, the user will be unaware that anything happened. This is not always possible; if the vanished mirror was the only one possessing certain pieces of state information, the element may restore itself to a state that doesn't exactly match its condition before the uncontrolled transfer. In this section, we discuss some of the state information that can be lost, and explore how to recreate it, or create a “safe” state that will allow the system to continue.
In addition to the problems specific to uncontrolled transfers, all the sorts of state synchronization problems mentioned in sections 7.2.2 and 7.2.3 for controlled transfers also apply here. Here, we deal with problems unique to uncontrolled transfers and their aftermath.

8.5.1. Recreating lost state

During controlled transfers, the source mirror transfers its current state to the destination. This is not an option during uncontrolled transfers. Instead, the new master mirror needs to create its internal state anew – perhaps based on partial or dated information, and perhaps from scratch.

**Ongoing computations:** If the vanished site was involved in an ongoing computation of some sort – an AI activity, perhaps – those ongoing computations are quite possibly gone for good. If the new master is a backup of the old one, it will have some information about what the vanished mirror was doing. It may be able to pick up where the vanished site left off, but first it needs to decide if the state information it has is still relevant.

Unfortunately, that determination is very specific to the element's particular task. An AI might check if its goals are still valid – if the destination it was planning a course for is still useful, or if the obstacle space it was using for calculations is unchanged. A game opponent might check if the player the old master was pursuing is still visible. If the goals are obsolete, or if a slave took over the master role and didn't know what those goals were, the AI will have to evaluate the environment and select new goals.

Other kinds of elements have fewer choices, but might suffer worse disruptions. A physics element, for example, might lose some information about ongoing changes to entities – current velocities, for example. If the backup mirror was being updated with information about the forces applied to objects, it might be able to recreate that information. But if the backup information is fairly old – or there was no backup, and only a basic slave was able to take over – it may have to start over from scratch, create a jarring (though temporary) discontinuity for observers.
Ongoing PG member calculations: If the element recovering from an uncontrolled transfer uses the M/PG/S distribution pattern, this creates both new opportunities and new challenges. A newly-promoted master mirror might be able to recreate more of its predecessor's state by querying the PG member mirrors, and learning what computations had been assigned to them. As with master-based calculations described above, there is still the chance that the PG members' computations are no longer relevant or meaningful, and this must be evaluated.

First, though, the new master must find out who its PG member sites are. A master will share that information with its backup mirror. Even then, the PG members need to be verified (since it's easy for multiple sites to leave the network at once). One of the first steps an M/PG/S master takes after an uncontrolled transfer is to broadcast a request for its PG members to identify themselves (and perhaps also, to describe what computations they were working on). This allows the newly-promoted master to discover its preexisting PG members, and lets it determine whether it needs to start searching for new mirrors to join the PG group.

Ongoing interactions: Perhaps the trickiest case to deal with is an uncontrolled transfer of an element that was involved in an interaction with some other element. In this case, the new master must take particular care when reconstructing the element's state, or it risks confusing the elements that it is interacting with as well.

Our manipulator and target example shows the problems that can occur. Imagine that the Manipulator:Target on some entity undergoes an uncontrolled transfer. When a manipulation begins or ends, the master informs all its slaves so that they can generate the results of a manipulation independently without requiring any additional communication. This means that, most of the time, any given slave knows about any ongoing manipulation; if it gets promoted to master and carries on from its own state as if nothing happened, the system behaves correctly.

The problem arises with edge conditions, where mirrors disagree about whether a manipulation is ongoing. If the master vanishes immediately after a manipulation begins or ends,
some of its mirrors might receive a manipulation update and some might not. This is especially likely if the old master vanished due to a networking failure.

To resolve this and get all mirrors on the same track, when the new master announced itself it can include in its message information about whether a manipulation is or is not ongoing. Given the circumstances, the new master itself may or may not have correct information to give out. There are two cases that can occur where the new master was unaware of a manipulation beginning or ending:

1. The vanished master had accepted a new manipulation, but had not informed all its slaves. The new master might be unaware that it should be dealing with an ongoing manipulation. For some applications (including the Datura sample applications), this situation can be ignored. The user's manipulation of an entity is interrupted, but he can still pick it up again. This is less safe when semiautonomous entities (AIs, for example) are doing the manipulating. They may not realize that a manipulation was disrupted, and become increasingly confused.

   There are a few ways to account for this. Remember from our discussion in Chapter 6 that, if the master mirror of a manipulator has gotten a response to its manipulation request, that response had to go through its local mirror of the ManipulatorTarget. So, that one ManipulatorTarget mirror does know about the accepted manipulation. When it receives an announcement of the new master it can send a message back to the master informing it about the previously accepted manipulation.

2. A manipulation had finished just as the master vanished, or during succession negotiations, and the new master mirror never received an update announcing the completion. This is basically the inverse of case 1. It can be handled in the same way, with the ManipulatorTarget mirror at the same site as the
ManipulatorElement master checking the manipulation state that’s part of the new master message and sending the master a correction if necessary. Alternately, as in the sample implementation, the master ManipulatorTarget can ask the manipulator element if the manipulation is still ongoing.

While it is always possible for state information to be lost when a master site vanishes, careful checking of reconstructed state can help a new master recover sanely and intelligently from an uncontrolled transfer. The most important goals, as demonstrated above, are to restore consistency between the mirrors of an element, and to avoid confusion when interacting with other elements. Checking information against other mirrors and even other elements can help the new master’s reconstructed state to closely match its predecessor's.

8.5.2. Reconciling multiple masters

We noticed above a rare condition where, due to a temporary split in the CVE network, our succession negotiation strategy could give rise to multiple master mirrors for an element. We also noted how the priority mechanism would cope, ensuring that one both masters are on a connected network one of them can take over. But what happens to the state of the “usurper” master that is overridden by the higher priority one? Ultimately, this is up to the design of the particular element. We can demonstrate how to handle different kinds of internal state using our sample set of elements.

Conflicting manipulations: If users on either side of the split were manipulating an entity – moving it around, for example – where should it be once the split is repaired? The trivial solution is to accept whatever position the prevailing master had. Another option might be to use whichever position is the result of the most recent manipulation, in hopes of making the discrepancy less noticeable to users.

Ongoing manipulations: A slightly more complex case is handling ongoing manipulations. If only one user is currently manipulating an entity, the prevailing ManipulatorTarget master should let the manipulation continue, even if the other, overridden, master was the one that permitted
it in the first place. To do this, it must acquire information about the overridden master's ongoing manipulations, make a decision, and send that decision to all of its mirrors.

If users on both sides of the split were manipulating the same entity when the split was repaired, the options are more limited. Many manipulation-handling elements (such as Datura's GrabbableElement and ResizeableElement) only accept one manipulator at a time, so the prevailing master has to interrupt one of the users and cancel his/her manipulation. Even if the handler can deal with multiple manipulators (Datura's MultiGrabbableElement is one such), the result of suddenly making both users simultaneous manipulators of one entity could be confusing.

Computations: If the element performs some ongoing computation, it may be useful to check if the overridden master had created any useful results that can be adopted by the new master. On the other hand, the element designer may consider this too much complication to benefit from an uncommon situation, and simply disregard any of the overridden mirror's internal state.

Processing Group members: Following a net split, an M/PG/S element will not only create dual masters on each side of the split, but those masters will also create new sets of PG members. If the split is later corrected, there will be certain mirrors that think they are PG members (of the overridden master) that the prevailing master is unaware of. The solution for finding PG members in section 8.5.1 deals with this situation, discovering all PG member mirrors and updating the prevailing master about their computations. A minor complication is that, in addition to receiving responses based on computations that the master considers obsolete, it might also receive responses for computation requests that it never sent.

8.6. Concrete examples

Before proceeding, we pause to explore some concrete examples of the concepts presented so far. In Datura's sample implementation, the support for uncontrolled transfers can be considered in three major pieces: assignment of succession priority numbers, actual succession negotiation, and state recovery/regeneration.
8.6.1. Assignment of priority numbers

Succession priority numbers are assigned through a simple message exchange between a newly-created mirror and the current master mirror of a particular element. The new mirror’s priority number is guaranteed to be unique.

In order to make this guarantee, the class implementing the priority number includes more than just the number itself. It also stores information about what child priorities it has previously given out to new mirrors. When a priority number is transferred between sites during controlled transfers, this extra information is included as well.

8.6.2. Succession negotiations

The process of succession negotiations begins when any mirror believes its current master has gone missing. It then calls its initializeSlaveWithoutMasterRole() method. In this example, we’ll use the implementation of that method in the BasicMSElement class. This is the base class of all elements that use the Master/Slave distribution pattern.

```cpp
Virtual void BasicMSElement::initializeSlaveWithoutMasterRole()
{
  if (isTransferable())
  {
    // this timer controls how long we wait before taking over
    // the Master role ourselves.
    mNewMasterWatchdogTimer.setMsecs (mNewMasterWatchdogInterval *
      (mSuccessionPriority.getEstimatedChildNum()));
  }
  mRole = SLAVE_WITHOUT_MASTER;
}
```

After that, the mirror broadcasts a "Missing Master" message (unless it had just received one). Sites that receive the Missing Master message respond in one of two ways, as noted in this code:
void BasicMSElement::receiveMissingMasterMessage ()
{
  if (mRole == MASTER)
  {
    // some site may have missed a heartbeat message, or
    // communications may have been interrupted. Send out
    // a reassurance to head off any succession negotiations.
    sendNewMaster ();
  }
  else
  {
    initializeSlaveWithoutMasterRole ();
  }
}

If the site that sent out the Missing Master message was mistaken, it will quickly receive a
correction from the master. Otherwise, all mirrors will now be in the Slave Without Master role, and
each will have a timer set based on how far down the line of succession it estimates itself to be.
These timers will themselves be slightly out of sync because of the time needed for the message to
propagate to the various sites; the default implementation used a fairly large (1 second) constant
multiplier on the timer values to provide a margin of error.

The timers are checked for each mirror during the element's update() call:

... if (mRole == SLAVE_WITHOUT_MASTER &&
    mNewMasterWatchdogTimer.expired ())
  {
    // Timer has run out. Time for this mirror to assume control.
    initializeMasterRole();
  }
...

Eventually, the timer at one of these sites expires and the mirror there assumes control and
notifies everyone else:

void BasicMSElement::initializeMasterRole ()
{
  mRole = MASTER;
}
mCurrentMaster = getSiteName();
sendNewMaster (mCurrentMaster, mSuccessionPriority);
}

A final bit of error checking happens when each site receives the new master message sent out in the above code:

```cpp
void BasicMSElement::receiveNewMaster (site_name, priority) {
    if ((mRole == MASTER) && (mSuccessionPriority > priority))
    {
        // Someone has incorrectly declared themselves master.
        // We need to take over. The easiest way to do this is
        // to go through all the steps we took when we first
        // became the master.
        initializeMasterRole();
    }
    else
    {
      initializeSlaveRole();
      mCurrentMaster = site_name;
      mLastKnownMasterPriority = priority;
    }
}
```

The tests in `receiveNewMaster` are designed to deal with a number of error conditions. First, there is a possibility that the highest-priority mirror was not the first to declare itself master. This could happen if, for example, the missing master message took a long time to get to it. Thus, the test here allows the highest priority mirror to correct any mistake and put itself back in charge.

This is also useful when dealing with network splits and reconnects. Whenever a master mirror of some element receives a message that only the master should have sent, it only needs to call `sendNewMaster()`. The "other master" will either accept the first master or send out its own New Master message with a higher priority. In either case, the lower-priority master surrenders its role to the higher priority one.
Once all sites have received a New Master message from the highest-priority surviving mirror, succession negotiations are completed. However, reconstructing the element's state may be an ongoing process.

8.6.3. State reconstruction

To consider how elements in the Datura sample implementation reconstruct their internal state after an uncontrolled transfer, we'll again use the example of the ManipulatorTarget element. This element uses the Master/Slave distribution pattern, so it inherits all the code we've looked at from the BasicMSElement class. We considered this element at the conceptual level in section 8.5.1; here we show the sample implementation's usage for comparison.

When a ManipulatorTarget slave is promoted to master, it has some initial state information to work off of. For example, it will know whether or not a manipulation was going on when the old master mirror vanished. However, it does not know if that information is current - the user may have released the entity after the old master vanished.

To check for this possibility, the ManipulatorTarget class appends some functionality to its initializeMasterRole method:

```cpp
virtual void ManipulatorTarget::initializeMasterRole ()
{
    BasicMSBehavior::initializeMasterRole ();
    if (mCurrentManipulator != NULL)
    {
        mCurrentManipulator->requestManipulationVerification ();
    }
    else
    {
        // Communicate to our slaves and manipulation-handling behaviors that no manipulation is happening anymore.
        requestRemoveManipulator (NULL, mActiveTarget, mLastPosition);
    }
}
```
If it thinks a manipulation was ongoing, it asks the manipulator in question whether or not that is still the case. The manipulator may need to query its own master mirror to get a definitive answer, so a slight delay is involved here.

Once the manipulator has an answer, it calls the ManipulatorTarget's manipulationVerificationResponse() method:

```cpp
virtual void ManipulatorTarget::manipulationVerificationResponse
    (ManipulatorInterface* mi,
     bool still_active,
     const gmtl::Matrix44f& final_position)
{
    if ((mCurrentManipulator == mi) && (!still_active))
    {
       requestRemoveManipulator (mi, mActiveTarget, position);
    }
}
```

RequestRemoveManipulator() is the method the manipulator would have called if the master had been available. Since the manipulator has informed us that the user has since released this entity, we simply call that method now. We also received a final position from the manipulator, which is used to provide a consistent final location for this entity on all sites.

8.7. Performance impact of uncontrolled transfers

Uncontrolled transfers affect the overall performance of a CVE in a variety of ways. Indeed, they are usually coincident with significant changes in the membership and network topology of the CVE itself. In this section we examine how this affects the performance of both the individual element undertaking an uncontrolled transfer, and the CVE as a whole.

8.7.1. Transfer design and unavailable time

As described in section 7.3.4, a significant concern when transferring a role is the unavailable time – the time during which no mirror has the Master role. In uncontrolled transfers, the unavailable
time is usually longer, and much more variable, than during controlled transfers. It is the sum of a
number of elements:

- The time required to detect the vanished site. This depends largely on how the
disappearance is detected. It might involve waiting for a message to timeout
without receiving a response – the timeout value needs to be configured with
some idea of the latencies to be expected in the system, with the proviso that
higher timeouts may be more noticeable to users. Of course, this only happens if
another mirror is expecting a reply from the master. Slower, but surer, is the
heartbeat method. Here there is a trade off between quickly detecting missing
sites and filling up available bandwidth with heartbeat messages. The sample
implementation, for example, defaults to a five second heartbeat, and raises the
alarm if 10 seconds pass without one.

- The time required to spread the notification of a missing master. A heartbeat
failure should be detectable on all sites, but otherwise the mirror detecting a
vanished master needs to inform its fellows. A mirror can't participate in
succession negotiation until it is made aware of the situation, after all.

- The time required until some site declares itself master. This is largely
dependent on the wait period calculated by each site. If the space of succession
priority numbers is fairly flat, this is a straightforward computation – a mirror
can tell it's nth in line of succession, so it waits n short intervals before assuming
control. The interval itself needs to be long enough that sites jumping ahead in
line are uncommon (though not impossible), which again depends on the latency
involved in getting messages across the network.

If several uncontrolled transfers occur, and a significant number of new
mirrors are granted priority numbers after each transfer, the space of assigned
priority numbers can grow more “tree-like”. Estimating the correct wait period requires estimating the number of higher-priority mirrors, which requires estimating the amount of branching in the priority number space. This can be managed by cautious estimates, and by listening to the priority number assignments to other mirrors. One possibility to correct this, during moments of calm, is for the master to revoke old priorities and assign a new, flat set of priority values to its mirrors.

If a site waits too long before declaring itself, that increases the unavailable time unnecessarily. If it waits too short a time, this increases the likelihood of usurping the Master role from a higher priority site, which leads to a set of corrections that further delay the process of finally resolving the succession.

- The time required for all sites to receive notice of the new master. Until the new master is known, a slave can't make a request for interaction, a PG member can't send the results of a computation, and so forth.

Sometimes, the delays caused by succession negotiation can be directly experienced by the users of a CVE. For example, if a mirror was requesting to begin an interaction on behalf of a user, it has to wait for negotiations to be finished before it can send the request again to the new master. Alternately, it can reject the interaction, forcing the user to try again later. This is why techniques for proactively detecting and dealing with vanished sites, such as using a heartbeat, or so important.

While succession negotiations using the priority scheme may take significantly longer than a controlled transfer, they are also a rare phenomena. Moreover, they can often be detected and corrected before being directly noticeable to the user. The entire transfer system is based around the desire to operate quickly under the most common circumstances, and tolerate longer delays so long as they happen rarely.
8.7.2. Effects on latency and load balancing

Controlled transfers most often happen with a goal of improving the performance of the CVE in some way, such as by load balancing (see Chapter 6) or by positioning roles to reduce the user's perceived latency (Chapter 9). These performance concerns are not addressed by the uncontrolled transfer strategy presented in this chapter; its performance concern is with restoring an element to a consistent, controlled state as quickly as possible. It gives no regard to user-interaction latency, and little regard to load balancing (but see 8.4.5, above).

Once the uncontrolled transfer completes its goal, the element is once again under control. That means that the Location Managers on the sites of the CVE can once again consider it for controlled transfers to more optimal locations.

8.7.3. The big picture – multiple uncontrolled transfers

Uncontrolled transfers are generally the result of losing an entire site, along with all the mirrors hosted on it. More often than not, this means that several elements will lose their masters. In the scheme we have outlined in this chapter, that means several elements will be conducting succession negotiations simultaneously and independently of each other. There are several performance considerations we can make to ensure the process runs as smoothly as possible.

First, in an ideal situation, the highest priority mirror of each negotiating element will be able to take control almost immediately. If that mirror needs to decline the Master role (as in section 8.4.5), the process is drawn out as the element waits for lower priority mirrors to take control. If multiple elements are conducting negotiations simultaneously, a degenerate condition is possible: If all elements assign priority numbers to mirrors in the same pattern, all those elements will have their highest-priority mirrors on the same site. This can easily lead to the site becoming heavily loaded and needing to decline promoting itself to Master for some elements. Worse, the process could repeat at the second-highest priority site, and so on down the chain of succession priorities until a site is found with free resources to handle the last, unluckiest element. The situation is illustrated in figure 8.1.
Figure 8.1. Assignment of succession priorities. In the top picture, all the mirrors at a particular site have the same priority; if Site A vanishes, each element will try to make its mirror at Site B the new master. In the bottom picture, priority assignments have been somewhat randomized. If Site A vanishes, each element has a different site hosting its highest-priority mirror.
This situation is easy to set up with a naïve algorithm for assigning priorities. In a common use pattern for CVEs, one site will create many of the initial entities in the environment. Other sites will join it one by one. If the primordial masters assign priorities in a strict sequence, all the mirrors on the first joining site will get '0:1', all the mirrors on the second site will get '0:2', and so on. Controlled transfers can ameliorate this somewhat, as priorities are swapped between the sender and receiver of a transfer, but there is still likely to be heavy pressure on individual sites.

A possible solution is to partly randomize the priority numbers being sent out. For example, the master could choose to assign any given mirror a randomly chosen, previously unused priority between '0:1' and '0:10' (and using a new range once this one is exhausted). That way, the highest-priority mirrors for different elements will be spread out among the different sites of the CVE, avoiding the heavy demand on a single site when a set of elements go into an uncontrolled transfer.

Some caution must be taken with the range of values, since priority numbers are also used to calculate the time a mirror waits before making itself a master. For example, if there's only one other site, and we randomly assign it the priority '0:256', the only effect of our randomization is that it will take quite a while for the other site to recover if we vanish. One solution to this is to assign numbers out of small ranges, as suggested above. There is a trade off between selecting a small enough range that there won't be wasteful gaps of unassigned numbers, and one large enough that the high-priority mirrors will be spread around effectively. If the expected use and number of users for a particular run of a CVE application can be estimated, this can be used to select a well-tuned range.

At any rate, if priority numbers are randomly, or partially randomly, distributed among the sites of the CVE, then multiple uncontrolled transfers can happen simultaneously with a low risk of interfering with each other. While this doesn't provide real resource usage-based load balancing, it also does not significantly burden any one site more than any others, and once the uncontrolled transfers conclude, controlled transfers can even out the load for any overly burdened systems. This also demonstrates why, in this situation, there is value in having the individual mirrors handle their
own succession negotiations, instead of having the Location Managers on all the other sites attempting to handle the entire succession process themselves.

8.8. Summary

The information in this and the preceding chapter demonstrated how transfers of role (and, along with roles, of computations) can be accomplished in a CVE under a variety of circumstances, both under control of the Location Managers in the system and independently in order to recover from errors and vanished sites. We also explored the particular performance concerns, both for the individual elements in the system, and also how these transfers impact interaction between elements of the CVE as a whole.

So far, the determining factor for controlled transfers has been the load-balancing method outlined in Chapter 6. However, there are other factors that can be considered by the Location Managers to ensure the best operation and interactivity of a CVE. Some of these, and their impact on system performance and the user experience, are described in the next chapter.
9. User-centric controls of Location of Computation

We have examined in depth how Datura performs transfers of location, both under organized control and as a response to disruptions in the system. We have described a Location Manager capable of acquiring various kinds of data and making LoC decisions, but so far have only described a basic load-balancing algorithm for its decision-making process. These capabilities correspond partially to two of our goals for LoC control: managing system load and providing error recovery.

This chapter expands on these goals and considers the third: improving user perception of system performance and latencies. We describe how the decision-making process in Datura's Location Manager takes advantage of user-centric information, and integrate that with our earlier accomplishments.

9.1. User considerations for selection policies

In traditional distributed computing, the algorithms used for deciding which computations to migrate to other sites are known as selection policies [Murthy/Manimaran01]. The load balancing described in Chapter 6 is a trivial example. The strategy used in that initial implementation was to simply select elements round-robin until the expected resource savings equaled the amount the system was overburdened. There are, of course, additional considerations that could be made. Traditional load-balancing typically considers the expected transfer time for moving a computation, and often looks for a best-fit solution – the smallest number of transfers that would meet the performance goal. Both of these can be applied in some form to a collaborative VR system, and turn up in the evolving strategies considered below.

CVEs also have their own special concerns, based around the needs and perceptions of their users. In developing selection policy, the system should choose transfers that have the least impact on the users' experience of and interaction with the environment. For example, an entity that the user is reaching for is a poor choice to undergo any transfers.
9.1.1. Evaluating candidates with multiple criteria

A selection policy can consider a number of very different criteria when choosing elements to transfer. In all cases the goal is to produce an element or set of elements that are the best potential candidates for transferring roles to other sites. This can be done by creating a score for each transferable element. Elements with the highest scores are the first ones chosen as transfer candidates.

Each criteria can be evaluated as a numeric value; these are then summed to create an element's final score. Weighting factors can be applied to these evaluations to scale them to common ranges, or to indicate the relative importance of different criteria. For example, a score might be calculated as being 80% based on processor load savings, and 20% on the transfer's likelihood of disrupting the user.

The trivial load balancer in Chapter 6 can be thought of as using this idea as well. In that case, elements that provided any possible savings (i.e. that had the Master or Processing Group roles) were given a score of 1.0, while slave elements received a score of 0.0.

Another issue when using multiple criteria to select elements for transfer is deciding when to make a transfer. The possibilities include:

- **When resource usage hits a high threshold.** In this case, the selector chooses enough of the highest-scored elements that its anticipated resource savings will push usage below the heavy load threshold.

- **When a user enters a new area of the environment.** As the user navigates through the virtual world, the set of entities with which he is likely to interact changes. This can make some entities' elements better choices to transfer to another site. It may also make some elements better candidates for transferring to that user's site – an idea we consider in section 9.3.
- **Periodically at regular intervals.** As a catchall, we can evaluate elements periodically. Unlike when evaluation is triggered by excessive resource usage, in this case the goal is not necessarily to select a certain number of transfers, but rather to look for any elements that might have an extremely high score. Any elements with a score beyond this cutoff become transfer candidates as a precautionary measure.

Datura's default behavior uses the first and last of these; performing selection based on the user's movement within the environment is more readily handled at the individual application level.

The remainder of this section considers particular criteria that can be used in calculating a score value. Section 9.4 describes the choices made in the Datura sample implementation: which criteria are used, and their relative weights.

### 9.1.2. Resource usage savings

The most obvious criteria for evaluating elements is the potential savings in resource usage if a role is transferred to another site. Elements can estimate their particular savings based on their current load and current role (versus the slave role they assume after a transfer).

An interesting point about resource usage savings is that the weight factor they are given in computing the score may change depending on the site's overall load. For example, if the site is very lightly loaded, the potential savings are of little concern. On the other hand, if the site is overloaded, the potential savings become a very important factor for scoring a particular element.

### 9.1.3. User proximity

We have seen in previous chapters that transferring roles has the potential to disrupt user interactions with entities in the CVE. For example, users cannot begin or end manipulation of an entity during the unavailable time when a `ManipulatorTarget` is being transferred from one site to another.
One easy way to quantify this is to prefer transferring elements that aren't in the user's immediate vicinity. For example, we can translate the entity's distance from the user into a value added to its score. This way, elements of farther-away entities are preferred for transfers\textsuperscript{18}.

In using a feature like user proximity, the designer needs to consider whether all users are treated equally. The policy could only consider users local to a particular site. On the other hand, it could consider all users it is aware of (for example, by collecting all entities with AvatarElements). The latter might allow better cooperation, but it diminishes the effectiveness of locality if there are a large number of users evenly spread through the space of the virtual environment. Alternately, remote users could be considered, but with less weight than the local users at a site.

9.1.4. Transfer difficulty

Another criteria is the difficulty or cost of transferring a particular element's role from one site to another. One quantifiable aspect of this is the amount of state information that would have to be sent in the transfer message. An element should be able to estimate this amount fairly easily. This is one of the components of the unavailable time we discussed in chapters 7 and 8, but not the whole thing: at this evaluation stage we can not know where a role would be transferred to. Those considerations come into play later in the process.

The existence of a backup for a role plays into this calculation indirectly. A recently-updated backup reduces the amount of data that must be sent in a transfer; thus, in general, mirrors with backups for their roles are favored over those without, and mirrors with up-to-date backups are favored over those less recently updated. Of course, this assumes that the role is transferred to its backup site. This is something we attempt to ensure in section 9.2.

\textsuperscript{18}In practice, it's useful to have a maximum amount that distance can contribute to the score. Otherwise, the distance of a very far away object will completely overwhelm other factors in the selection process. For example, one test implementation of this feature capped distances at 100 meters, and then scaled the result to a number in [1.0,3.0].
There are other factors that could be taken into consideration under the general label of "transfer difficulty". For example, some elements need to reconstruct large amounts of data after a transfer. An element could provide particular hints that are summed into its total score.

9.2. User considerations for location policies

Location policies are the other half of distributed system decision making. Where selection policies decide which computations to transfer, location policies determine where to send them. For controlled transfers in Datura, the location policy comes into play after sending out a request for bids. It determines how long we wait to collect bids from different sites, and how we choose which bids to accept.

Given a set of bids from a number of sites, a variety of criteria affect the selection of the "best" one. These choices are combined via scoring, similarly to how the selection policy works. The criteria include:

9.2.1. Backup mirrors

Backup mirrors make the choice obvious – they exist specifically as a predetermined place to transfer a mirror's role to. Bids from any other site should only be considered under very extreme circumstances – e.g. if the network connection to the backup site is very poor, or if it doesn't respond with a bid at all. In terms of scoring, bids from a backup mirror's site should receive a very large value.

9.2.2. Estimated unavailable time

The selection strategy did not have enough information to determine the unavailable time for an element undergoing transfer, but at this stage it may be possible to make a reasonable estimate. If the underlying networking implementation gives us enough information about latency and bandwidth measurements between sites, then we can estimate the unavailable time incurred by accepting a particular bid.
Of course, in our scoring system, lower unavailable time estimates should result in a higher score.

9.2.3. User proximity

In selecting elements for transfer, we gave preference to those that are far away from users. Once the decision has been made to transfer a particular mirror's role from one site to another, it may be possible to choose a destination site that is advantageous to at least some users. User manipulation of entities is best (lowest latency and least uncertainty) when the Master role of the ManipulatorTarget is on the same site as the user. Therefore, it can be helpful to look for users close to the entity undergoing the transfer, and give preference to the sites hosting those users.

For example, if some element of a "chair" entity is being transferred, and only one user is near that chair, the site hosting that user will be preferred by the location policy. If that user tries to move the chair, the manipulation can begin immediately.

9.3. Seeking work

Traditional load balancing is generally concerned with offloading excess work. While sites obviously can accept work when lightly loaded, and may even advertise their availability, it is rare for them to actually seek out particular computations that they actively want to host. However, the tools the Datura model provides may make that a viable option.

Generally, the only reason a particular site would seek to take on an extra workload is if doing so would directly benefit the user or users at that site. This might happen for entities that the user is expected to interact with, so as to improve interaction latency. However, this is countered by the fact that we do not want to have a role transfer in progress when the user actually does try to interact with the entity. A clever LM policy could seek transfers to its site when the user enters a new area in the virtual environment, transferring elements that are "close, but not too close" to the user.
9.4. Implementation considerations

The Datura sample implementation, used for the testing and evaluation experiments described in Chapter 10, makes specific choices about which of these factors to use and how to weight them.

Selection policy: Load balancing is based primarily on CPU usage, and is tested using a combination of threshold checking and periodic reevaluations after the threshold is exceeded (Section 9.1.1). Selection of elements to transfer is based primarily on expected savings (Section 9.1.2), and the weight is increased during heavy loads. A secondary concern is the difficulty of transfer, as rated by the element in question. A lower weight is given to locality of users (Section 9.1.3); this criteria is basically used to break ties between similarly-performing elements. Remote users contribute 25% as much to the score as local users.

In terms of the final score, computation savings typically contribute a number in the range of 0 to 10 (0 to 20 under very high loads), transfer difficulty 0 to 5, and locality 0 to 3.

Location policy: Destinations for transferred elements are selected primarily based on latency of connections – or more specifically, how quickly bidding sites respond. The locations of the users hosted by potential destination sites are given a lower weight. User locality is used as a tiebreaker among bids that arrive in a given interval. Backup mirrors are given a large fixed bonus to their scores to prioritize them over other bidders (Section 9.2.1).

The sample implementation sites do not explicitly seek to acquire specific roles or authority.
10. Testing and evaluation

In Chapter 4, we introduced the Datura sample implementation, a software library implementing a rich set of elements and demonstrating many of the LoC techniques described in the previous chapters. This implementation was used for developing and testing the LoC methodologies. In this chapter a specific set of experiments are described.

The experiments serve two purposes: first, to verify correctness of the Datura system; and second, to examine the relation of location management to performance. The specific goals of the testing process are outlined below:

Correctness goals

- Verify correct distribution of elements. We must ensure that the Datura system correctly creates mirrors of all elements at each site. We must also test how roles are assigned to elements; for example, checking that each M/S element has exactly one Master mirror, and verifying that an M/PG/S element builds up a set of processor sites as expected.
- Verify state transfer/reconstruction during transfers. We must test elements during controlled and uncontrolled transfers of roles, and ensure that the elements act correctly before and after the transfer.
- Verify correct recovery from lost sites via uncontrolled transfers. Demonstrate benefits of distribution patterns that use backup nodes.

Performance goals

- Measure how latency of user interactions with objects in the environment varies depending on which site decisions are made at.
- Test the load balancing algorithm. Demonstrate how elements are selected for transferring under various circumstances, and measure the time required to
perform transfers.

- Measure unavailable time for elements caused by uncontrolled transfers.
- Demonstrate performance benefits of M/PG/S distribution pattern.

Ten specific tests were formulated based on these goals; these are described and the results analyzed below. As a matter of practicality, a standard environment – both the physical environment of sites and network connections as well as the virtual environment represented in the application – was selected for use in all of the tests.

10.1. Sites and network layout

The standard test network consists of five heterogeneous sites, connected as shown in Figure 10.1. Four of the sites were physically close to each other, so that they could be easily monitored and interacted with, while the last is situated remotely to demonstrate the effects of real-world network lag.

![Site specifications]

1. Dual-processor Linux
2. Single-processor Linux
3. Multiprocessor Irix
4. Dual-processor Linux
5. Dual-processor Linux

Figure 10.1. Layout of test network.

The sample implementation uses the Plexus network library, which overlays a virtual message routing algorithm over an existing network. For example, messages sent from site 1 to site 5 in the
test network always travel by way of site 4. Thus, these messages have a correspondingly higher latency than messages from site 1 to site 4 or site 3 – as will be demonstrated in our second major test case.

10.2. Contents and initial distribution of the virtual environment

The initial library of elements for Datura were selected to meet the requirements of the VR Juggler Template Applications; those elements were enumerated in section 4.1.1. The more interesting distribution patterns were not represented in this set, and so a more elaborate application was developed for the experiments.

The Robot Playground application, first publicly shown in 2000, was an experimental application for Datura's predecessor, Octopus. That application's special functionality – groups of autonomous robots that interacted with users and the environment – was refactored as a set of Datura elements. A combined environment was created from elements of Robot Playground and the Template Applications; this was populated with a variety of omnipresent entities. Figure 10.2 illustrates the complete environment.

Figure 10.2. Wide-angle view of the combined Template App / Robot Playground.
10.2.1. Objects in the virtual environment

The environment consists of the following kinds of objects, each represented in software as an omnipresent entity composed of elements:

**Terrain.** The "marketplace" model provides context and landmarks for human users. Rectangular "arenas" are used to constrain movement of the robots – they stay within their arena and interact only with the objects inside of it. These entities use a small set of elements: PositionReceptor, ModelInfoReceptor, SizeReceptor, and ViewableElement.

**Movable pieces.** There are several kinds of "furnishings" that users of the application can interact with. These include clay pots that were modeled to fit in with the architecture of the marketplace model, and resizable obstacle blocks that can be used to create challenges for the pathfinding robots described below. There are also two special objects for use by the pathfinder robots: a "cargo" object, which they can pick up, and a "landing pad" (the pathfinders' goal is to deliver the cargo to the pad's location). The omnipresent entities representing these objects are composed of PositionReceptors, ModelInfoReceptors, and ViewableElements. Interactivity is provided by a ManipulatorTarget and GrabbableElement. Additionally, the resizable obstacle blocks use SizeReceptors and ResizableElements.

**Pathfinder robots ("hueys").** The default configuration places three pathfinder robots in the central arena (see Figure 10.4.). Each robot is designed to perform the same task: When triggered by any user, they try to pick up a cargo object and deliver it to a target landing pad. Each of the two stages of movement (to the cargo, and then to the target) requires a pathfinding computation which the robot uses to avoid obstacles. Obstacles are any entities in the environment that have SizeReceptors attached to them. The robots are also constrained to acting within a 10x10x3 meter arena.
Figure 10.3. A group of movable objects: clay pots to fit the “marketplace” motif.

The pathfinders all use a randomized roadmap algorithm for path planning, in which random points are created in the available space, and connections made between points if the robot can travel between them without colliding with other objects. They differ in the distribution pattern used for the element that handles the robot AI and path planning: one uses the basic M/S pattern, the second uses M/B/S (M/S with a backup mirror), and the third uses M/PG/S. In the tests, these will be used to compare capabilities of the various patterns.

All told, each pathfinder requires a significant set of elements: PositionReceptor, ViewableElement, ManipulatorTarget (so that users can place them in the environment and turn them on and off), GrabbableElement, and PathfinderElement (the AI control element, one of three varieties with different distribution patterns).
Figure 10.4. Three "huey" pathfinding robots, each using a different distribution pattern for its AI element. Also visible are the cargo object (lower right) and the target pad (lower left). Several resizable blocks are also visible in the arena.

**Gravbot robots.** A second group of robots are contained in a small arena to the north of the pathfinder robots' arena (Figure 10.5.). Unlike the pathfinder hueys, the gravbots are always on. In the absence of user input, they wander aimlessly through their arena, bouncing off of obstacles. Users can also attract the gravbots to their hands.

The gravbots create a small constant load on whichever site hosts the Master mirrors of their controlling behaviors. They also demonstrate the use of interpolation to smooth out errors in movement on slave mirrors. Aside from the M/S `GravbotBehavior` element that controls movement, user interaction, and collision detection, gravbots only require a `PositionReceptor` and custom `ViewableElement` (they cannot be explicitly grabbed by users).
10.2.2. Object and user locations

The initial locations of objects in the virtual environment are set by the configuration files given to each site. Objects are located in four major areas, as noted in Figure 10.6. Most of the "action" of the tests happens in and around Area A; this is where users at each site appear when they join the CVE. Users will move to other areas as needed, either to manipulate objects in those locations or to evaluate the location-specific capabilities of the sample implementation's location management routines.

Of course, location means two things in a distributed network. The other meaning - the locations of primordial and master mirrors of elements among the sites joined to the CVE - is discussed in the first test in the next section.
10.3. Tests and results

This section describes ten basic tests of the Datura sample implementation's functionality. Each test begins with the five sites in the test network starting up in numerical order, each one adding a few of the objects described in section 10.2. Thus, each test starts from a common base point.

The initial load balancing parameters at each site were set to ensure all sites considered themselves lightly loaded (load thresholds were set to very high values). For tests of uncontrolled transfers, this meant that absolute statements could be made about the state of the CVE before and after removing a site (or sites). For tests of controlled transfers and load balancing itself, the parameters were reset to “interesting” values once the test was underway, so that the results of load balancing could be easily watched and logged.

Each test was performed multiple times; times reported are averages over multiple runs unless otherwise noted. Maps of role distributions among sites are generally of “typical” individual runs.
### 10.3.1. Test 1: Entity creation and steady-state measurements

The primary goal of the first test was simply to ensure that all entities were created correctly on all sites; a secondary goal was to collect baseline network usage and processing load information about each site.

"Created correctly" entails several specific requirements:

- Every omnipresent entity, and every element thereof, exists at every site.
- All the mirrors of each element have valid roles (e.g. only one master or backup).
- All mirrors that need them have valid and unique succession priority numbers.
- All M/B/S and M/PG/S elements are able to create backup mirrors and PG members, respectively.

To test these requirements, a role map was created of the CVE after all five sites joined. The role map simply lists, for each site, all extant elements, with their roles, priority numbers, and current processor usage (in ms per update). We illustrate the result with a short extract from the role map created for one of the test runs (to save space, only a few entities and elements are shown):

<table>
<thead>
<tr>
<th>ROLE MAP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sites:</td>
</tr>
<tr>
<td>site3</td>
</tr>
<tr>
<td>site1</td>
</tr>
<tr>
<td>site4</td>
</tr>
<tr>
<td>site5</td>
</tr>
<tr>
<td>site2</td>
</tr>
<tr>
<td>Total load (ms):</td>
</tr>
<tr>
<td>28.06</td>
</tr>
<tr>
<td>3.66</td>
</tr>
<tr>
<td>4.30</td>
</tr>
<tr>
<td>3.55</td>
</tr>
<tr>
<td>3.19</td>
</tr>
</tbody>
</table>

#### MSPathFinder

<table>
<thead>
<tr>
<th>ManipulatorTarget</th>
</tr>
</thead>
<tbody>
<tr>
<td>Role: Slave</td>
</tr>
<tr>
<td>Priority: 0a:0a:</td>
</tr>
<tr>
<td>Load (ms): 0.0047</td>
</tr>
</tbody>
</table>

#### MSPathFinderElement

| Role: Slave       |
| Priority: 0a:4a:  |
| Load (ms): 0.0017 |

#### Gravbot1

<table>
<thead>
<tr>
<th>SparkleViewableElement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Role: Peer</td>
</tr>
<tr>
<td>Load (ms): 0.0015</td>
</tr>
</tbody>
</table>

#### GravbotBehavior

| Role: Master         |
| Priority: 0a:        |
| Load (ms): 0.78      |

| Role: Slave         |
| Priority: 0a:3a:     |
| Load (ms): 0.24      |
| Role: Slave         |
| Priority: 0a:4a:     |
| Load (ms): 0.25      |
| Role: Slave         |
| Priority: 0a:2a:     |
| Load (ms): 0.26      |
| Role: Slave         |
| Priority: 0a:0a:     |
| Load (ms): 0.12      |
The results in the excerpt (and in the remainder of the role maps over multiple runs) bear out our expectations. Because the sites were configured to avoid load balancing at this stage, the primordial mirrors of all Master/Slave elements are the current master mirrors listed in the role map. Slaves are given succession priority numbers in a randomized way such as we described previously, but retain uniqueness among all mirrors of a given element.

The timing data provides some interesting information. For each element, the time required for a single update call is measured and averaged over several seconds. These are also summed to create the total load measurements at the top of the excerpt.

It is immediately apparent that Site 3 is much more heavily loaded than the other sites, and this is in fact the case. Site 3 is a multiprocessor Irix mainframe. While it has a large number of processors, each is individually slow compared to the CPUs in the Linux workstations, thus requiring much more total time to perform approximately the same amount of work. In fact, much of that load comes from updating the graphics of the user avatars (which involves applying transformation matrices to a large mesh of vertices). Note that this work is happening on slave mirrors – unfortunately, we can not load balance away this particular source of load.

Other elements show interesting results; both the TrackedElement on the Avatar and the GravbotBehavior on Gravbot1 show noticeably higher loads on their master mirrors than on their slaves. We've noted before that TrackedElements cannot reasonably be the subject of role
transfers, but when we come to the load balancing tests the gravbot robots will be fair game.

We can also note a few other comparisons about the speed of our test machines. Sites 1, 4, and 5 are essentially identically configured machines, though Site 4 suffers slightly because it hosts more active elements in our default configuration (it introduces the pathfinder robots, as well as their cargo and target objects, into the CVE). Site 2, the remote Linux workstation, is newer and slightly faster than the other three.

As part of Test 1, the message output of each site was also logged. Table 10.1 enumerates the messages sent from each site in a typical 10 second period.

<table>
<thead>
<tr>
<th>Site</th>
<th>Messages and sizes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Site 1</td>
<td>84 x 45 bytes + 2 x 17 bytes</td>
</tr>
<tr>
<td>Site 2</td>
<td>82 x 45 bytes + 2 x 17 bytes</td>
</tr>
<tr>
<td>Site 3</td>
<td>82 x 45 bytes + 2 x 17 bytes + 75 x 41 bytes</td>
</tr>
<tr>
<td>Site 4</td>
<td>86 x 45 bytes + 2 x 17 bytes</td>
</tr>
<tr>
<td>Site 5</td>
<td>82 x 45 bytes + 2 x 17 bytes</td>
</tr>
</tbody>
</table>

Table 10.1. Number and sizes of messages created by each site during a 10 second interval. Sizes are for message payloads, and do not include headers.

Unsurprisingly, in this quiescent state (with no user interactions occurring), most of the sites look essentially the same. The 45 byte messages are position updates sent by the TrackedElements that are used for user avatars, while the 17 byte messages are heartbeats sent by the Location Manager at each site. These messages are all sent at regular time intervals. The 41-byte messages from Site 3 are sent by the master mirrors of the gravbot's controlling elements; they send a course correction whenever the gravbot hits an obstacle. Of course, additional messages will be created when users interact with the CVE and sites are added or removed; we'll look at some of that additional network traffic in subsequent tests.
10.3.2. Test 2: Location impact on user interaction latency

When users interact with an object in the environment – picking something up, for example – a decision needs to be made about whether or not to allow the interaction. In the Datura model, this is handled by the master mirror of a ManipulatorTarget element. The user’s perceived latency depends on how long it takes to get permission (or a denial of permission) from that element. Test 2 examines how the relative locations (in the network) of the user and the permission granter affect the user’s perception of latency.

In this test, a user located at Site 1 repeatedly tried to grab objects whose ManipulatorTarget masters were located at each Site. The time required to send a request to the ManipulatorTarget master and receive a response is shown in Table 10.2.

<table>
<thead>
<tr>
<th>Circumstance</th>
<th>Average time (in ms)</th>
<th>Best-case time (in ms)</th>
<th>Worst-case time (in ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local decision – Site 1</td>
<td>0.21</td>
<td>0.16</td>
<td>0.27</td>
</tr>
<tr>
<td>LAN (1 hop) – Site 4</td>
<td>16.95</td>
<td>4.68</td>
<td>31.31</td>
</tr>
<tr>
<td>LAN (1 hop) – Site 3</td>
<td>11.45</td>
<td>5.53</td>
<td>32.64</td>
</tr>
<tr>
<td>LAN (2 hop) – Site 5</td>
<td>25.55</td>
<td>4.95</td>
<td>57.47</td>
</tr>
<tr>
<td>Internet – Site 2</td>
<td>140.31</td>
<td>76.81</td>
<td>255.11</td>
</tr>
</tbody>
</table>

Table 10.2. Latency times for initiating interactions.

Naturally, the best result occurs when the decision is made at the same site as the user. When the decision is made on a LAN connection with only one network “hop”, the results are very acceptable: responses often arrive in time to be accounted for when computing the next graphics frame of the application. When this happens, the network delay will be invisible to the user, overshadowed by the inherent delay of drawing a graphics frame.

When the messages have to make two trips across the LAN (in each direction), the average time required to get a response approximately doubles, often resulting in several frames of delay.

19The graphical frame rate of the application depends on a great many factors. 15 frames per second
before the user sees a response to his attempt to interact with an object. When the messages have to travel across the Internet, the situation deteriorates further. Our measured worst-case time is over a quarter of a second, obvious to any user, and it is easy to imagine worse results happening for farther-flung locales or more heavily saturated connections. Clearly, the location of decision-making relative to the location of the user can have a very obvious impact on the user's experience of the CVE.

10.3.3. Test 3: Pathfinder robot distribution and performance

The third test demonstrated the possible utility of the Master/Processing Group/Slave distribution pattern. To do this, two versions of the pathfinder robot were used; one using the M/S pattern for its pathfinding element, the other using M/PG/S. The third implementation (M/B/S) was omitted, since it acts identically to the M/S version with regards to the pathfinding algorithm.

The use of the processing group by the pathfinder was atypical of distributed computing; instead of dividing the problem into smaller pieces for each processor, in this implementation each PG member tried to solve the path planning problem independently. Since the pathfinding algorithm is randomized, the required time to find a valid path can vary significantly, even for the same problem. The time required to find a path is the time used by the "luckiest" PG member. Thus, using multiple PG members should reduce the average time needed to solve the path planning problem.

In the test, each robot performed ten test runs, with the same initial environment each time. The M/PG/S pathfinder was configured to look for up to three PG member sites. Because it did this while the individual sites were connecting to the CVE (always in the same order), Sites 1, 2, and 3 were always selected as its PG members. In this particular implementation, the master site for the M/PG/S robot did not run the pathfinder itself, and only acted as a coordinator for the PG members.

Table 10.3. summarizes the results of these tests. Times are measured from when the master mirror began searching, to when it found or received a solution. Thus, times include the

(67 ms per frame) is often used as a minimal standard of acceptability, while 30 fps (33 ms per frame) or higher
<table>
<thead>
<tr>
<th>No processing group</th>
<th>Stage 1</th>
<th>Interaction latency</th>
<th>Stage 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average time (ms)</td>
<td>12111.8</td>
<td>0.32</td>
<td>13371.5</td>
</tr>
<tr>
<td>Best case (ms)</td>
<td>4705.9</td>
<td>0.26</td>
<td>4179.6</td>
</tr>
<tr>
<td>Worst case (ms)</td>
<td>22369.8</td>
<td>0.48</td>
<td>25164.4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>With processing group</th>
<th>Stage 1</th>
<th>Interaction latency</th>
<th>Stage 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average time (ms)</td>
<td>7564.2</td>
<td>0.25</td>
<td>12817.7</td>
</tr>
<tr>
<td>Best case (ms)</td>
<td>3370.5</td>
<td>0.17</td>
<td>6835.6</td>
</tr>
<tr>
<td>Worst case (ms)</td>
<td>16999</td>
<td>0.31</td>
<td>29014.5</td>
</tr>
</tbody>
</table>

Table 10.3. Effects of distribution pattern on pathfinder performance. Two paths are found: Stage 1, from the robot’s initial position to the cargo object, and Stage 2, from the cargo to the landing pad. Interaction latency is the time required to grab the cargo once the robot reaches it.

As expected, the average times required for both stages are reduced when the processing group is used. The degree of improvement is very different between the two stages. Looking at the complete data, it appears that this was caused by several particularly ill-suited attempts by the randomized roadmap algorithm, including the one which generated the all-time worst case 29 second search time.

It is also worth considering the impact of the processing group approach on the overall load of the system. Table 10.4 displays average load information for each site with and without the processing group. Of course, this is partially under the control of the robot’s PathfinderElement; in this implementation, the pathfinding algorithm attempts to limit the amount of processing time it uses in a single update cycle, subject to the granularity of the algorithm itself. The much larger increase seen on Site 3 is indicative of the time required by the slower processor to reach a stopping point in the calculation, and highlights the problem of comparing loads across highly diverse sites. If the load balancing parameters were set more conservatively, Site 3 would not volunteer itself as a PG member at all.

is desirable. By comparison, American television runs at 30 fps, and motion pictures at 24 fps.
The remaining tests explore various aspects of Datura's location management capabilities, beginning with basic tests of controlled and uncontrolled transfers. These first several tests assume a mostly quiescent application: no ongoing manipulations, and no activity on the part of the pathfinder robots, although the gravbot robots are always active. The final several tests specifically explore the effects of role transfers on ongoing manipulations and the pathfinder robot's activities.

Test 4, then, observes what happens when the load balancing capabilities of the Datura Location Manager come into play. In order to create a controlled environment for gathering observations, the sites of the CVE were started up as usual – with very liberal allowances of time for updating elements. The load thresholds were then set to lower values at certain sites, ensuring that those sites would try to transfer Master and Processing Group roles to other sites to reduce their load.

There are several things to observe during the transfer of roles, including:

- How many elements are transferred, and in what order?
- Which sites' bids are accepted?
- What is the unavailable time for each transfer?

In the first set of trials, the heavy load threshold at Site 3 was set to 90% of that site's current load. This naturally caused a series of transfers as it sought to reduce its load. Table 10.5 summarizes the final results: the distribution of transferable roles across different sites before the load balancing parameters were changed, and again after the sites had returned to a steady state (we'll analyze the particular choices of which elements were chosen in the next section). These results are for a typical run.

### Table 10.4. Update times at each site during pathfinding. All times in milliseconds.

<table>
<thead>
<tr>
<th></th>
<th>Site 1</th>
<th>Site 2</th>
<th>Site 3</th>
<th>Site 4</th>
<th>Site 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>No processing group</td>
<td>3.79</td>
<td>3.20</td>
<td>29.15</td>
<td>11.24</td>
<td>3.65</td>
</tr>
<tr>
<td>With processing group</td>
<td>11.00</td>
<td>9.81</td>
<td>55.88</td>
<td>3.59</td>
<td>3.64</td>
</tr>
</tbody>
</table>

10.3.4. **Test 4: Basic load balancing and controlled transfer evaluation**
In attempting to reduce its load, Site 3 successfully transferred 8 Master roles and one PG Member role. Site 1 ended up as the most common recipient; it still had large amounts of free resources according to its load balancing settings, and by virtue of the network topology its bids would arrive at Site 3 before any other site’s.

The unavailable times for the elements that transferred their Master roles are most easily measured at Site 3; this site knows exactly when it gave up Master status for those elements, and can compare this to the time when it receives notification of the new master’s readiness. Other sites don’t notice anything happening until the new master notification is sent. During this experiment, the unavailable times for transferred roles ranged from 3.1 to 27.4 ms, averaging 10.8 ms.

For the second set of trials, the process was repeated with Site 4 (host of the pathfinder robots, among other things) as the overloaded site. Table 10.6. gives an example of how roles were distributed after the load balancing had reached a steady state. During this set of trials, the unavailable time ranged from 2.9 to 36.0 ms, averaging 15.5 ms. The slight average increase over the first set of tests may be due to the transfer of the Master role for the pathfinder robots’ control elements; these need slightly more reinitialization when a new master is created than some other elements.
Table 10.6. Role distribution before and after balancing, when site 4 becomes overloaded.

<table>
<thead>
<tr>
<th></th>
<th>Site 1</th>
<th>Site 2</th>
<th>Site 3</th>
<th>Site 4</th>
<th>Site 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Before LB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Master</td>
<td>8</td>
<td>13</td>
<td>15</td>
<td>16</td>
<td>32</td>
</tr>
<tr>
<td>PG Member</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Backup</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Slave</td>
<td>74</td>
<td>70</td>
<td>68</td>
<td>68</td>
<td>52</td>
</tr>
<tr>
<td>After LB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Master</td>
<td>14</td>
<td>13</td>
<td>15</td>
<td>7</td>
<td>35</td>
</tr>
<tr>
<td>PG Member</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Backup</td>
<td></td>
<td>1³⁰⁷</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Slave</td>
<td>70</td>
<td>69</td>
<td>68</td>
<td>77</td>
<td>48</td>
</tr>
</tbody>
</table>

As seen in the table, when Site 4 became overloaded and had to lighten its load, nine elements had their Master roles sent to other sites. These were divided between the nearest two sites: 1 and 5.

Of particular interest are the fates of the control elements for the M/PG/S and M/B/S versions of the pathfinder robots. Both of these happened to have their Master roles transferred from Site 4 to Site 1. Because of this, these elements had to allocate a new PG member and a new backup mirror, respectively.

While these tests display the viability of load balancing, and demonstrate that the Datura system can successfully transfer roles from one site to another, they also point to a possible limitation in the sample implementation’s accounting for resource usage. In this quiescent state, relatively little work is being done by most elements, and the difference in processing requirements for a master versus a slave is small. While the load balancing distributes responsibilities to less loaded sites, the real benefit of this is not fully visible until the elements whose roles were transferred are actually called upon to act or interact with users.

10.3.5. Test 5: Location influence on load balancing

In Test 4, all the users in the CVE were gathered around Area A (see Figure 10.6 for a map).

³⁰During the initial run of tests, an implementation error was discovered: When a backup mirror acquired the Master role via a controlled transfer, it would not recognize that it needed to appoint a replacement backup mirror. The tests were rerun once this glitch was corrected.
The location (in the virtual world) of users is considered by the Location Manager when choosing which elements to transfer for load balancing. The concept is to prefer against transferring roles for any elements located near users, with the goal of minimizing noticeable disruptions to users.

Test 5 illustrates the effect of user positioning. As with Test 4, we tighten the load balancing parameters at one site so that it will attempt to offload responsibilities to other sites. In this case, we use Site 5 for the experiment, since the set of objects it introduces (and is initially responsible for) is geographically distributed across the virtual environment. Site 5 introduces a set of resizable blocks and clay pots at Areas A, C, and D on the map. The experiment is repeated with the users moving from one area to another; Table 10.7 shows the order that elements are selected for role transfers.

<table>
<thead>
<tr>
<th>Users at Area A</th>
<th>Users at Area C</th>
<th>Users at Area D</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. areaDpot4:ManipulatorTarget</td>
<td>1. areaDpot5:ManipulatorTarget</td>
<td>1. areaBlock1:ManipulatorTarget</td>
</tr>
<tr>
<td>2. areaDpot1:ManipulatorTarget</td>
<td>2. areaDpot5:DummyLoad</td>
<td>2. areaDpot1:ManipulatorTarget</td>
</tr>
<tr>
<td>3. areaDpot5:ManipulatorTarget</td>
<td>3. areaDpot1:ManipulatorTarget</td>
<td>3. areaDpot1:DummyLoad</td>
</tr>
<tr>
<td>4. areaDpot2:ManipulatorTarget</td>
<td>4. areaDpot2:ManipulatorTarget</td>
<td>4. areaBlock2:ManipulatorTarget</td>
</tr>
<tr>
<td>5. areaDpot3:ManipulatorTarget</td>
<td>5. areaDpot2:ManipulatorTarget</td>
<td>5. areaDpot2:ManipulatorTarget</td>
</tr>
<tr>
<td>6. areaDpot5:DummyLoad</td>
<td>6. areaDpot4:ManipulatorTarget</td>
<td>6. areaDpot2:DummyLoad</td>
</tr>
<tr>
<td>7. areaDpot3:DummyLoad</td>
<td>7. areaDpol4:DummyLoad</td>
<td>7. areaBlock4:ManipulatorTarget</td>
</tr>
<tr>
<td>8. areaDpot2:DummyLoad</td>
<td>8. areaDpol2:DummyLoad</td>
<td>8. areaBlock3:ManipulatorTarget</td>
</tr>
<tr>
<td>9. areaDpot4:DummyLoad</td>
<td>9. areaDpot1:DummyLoad</td>
<td>9. areaDpol1:ManipulatorTarget</td>
</tr>
<tr>
<td>10. areaDpot1:DummyLoad</td>
<td>9. areaDpot1:DummyLoad</td>
<td>10. areaDpol1:DummyLoad</td>
</tr>
<tr>
<td>11. areaDpot1:ManipulatorTarget</td>
<td>10. areaDpot2:ManipulatorTarget</td>
<td>10. areaDpol1:DummyLoad</td>
</tr>
<tr>
<td>12. areaDpot3:DummyLoad</td>
<td>11. areaDpot2:DummyLoad</td>
<td>11. areaDpol2:ManipulatorTarget</td>
</tr>
<tr>
<td>13. areaDpot2:ManipulatorTarget</td>
<td>12. areaDpol3:ManipulatorTarget</td>
<td>12. areaDpol2:DummyLoad</td>
</tr>
<tr>
<td>14. areaDpot1:DummyLoad</td>
<td>13. areaDpot1:ManipulatorTarget</td>
<td>13. areaDpol3:ManipulatorTarget</td>
</tr>
<tr>
<td>15. areaDpot1:ManipulatorTarget</td>
<td>14. areaDpot1:DummyLoad</td>
<td>14. areaDpol3:DummyLoad</td>
</tr>
</tbody>
</table>

Table 10.7. The order that elements are selected for transfers depending on users' location in the environment. Elements are identified by the name of the Omnipresent Entity they are attached to, followed by the name of the C++ class implementing the element.

In the left-hand column, the test is run with the users of the environment gathered around Area A. When forced to reduce its load, the Location Manager for Site 5 begins with elements attached to omnipresent entities at Area D, far away from the users, and then chooses from the slightly closer Area C. The right hand column is the reverse case, with users at Area D; the preferred elements for transfers are then at Area A. When the users gather in the middle (center column), elements from
either side are chosen (though favoring the slightly farther away Area D).

10.3.6. Test 6: Basic uncontrolled transfer evaluation

As with the controlled transfer tests above, the testing of uncontrolled transfers begins with the application in a quiescent state. Test 6 involves removing a site from the network by killing the application running at that site. When the dead site fails to send heartbeat messages, the remaining sites go into error recovery.

In the first test runs, Site 3 was removed. Site 3 was the initial host of the gravbot robots and various resizable blocks located in Area B. When it is removed, all the slave mirrors of the gravbots' GravbotBehavior elements and the blocks' ManipulatorTarget elements find themselves without masters. The remaining sites must negotiate new masters for those elements. Also, the entities that make up the user's representation (avatars and manipulators) must be removed at the other sites. Finally, all the elements remaining in the system must end up in a usable state, consistent across different sites and able to respond correctly to user interaction.

An examination of the role distribution among the participating sites can give a good example of the big picture. Table 10.8 compares the distribution of roles before and after removing site 3. Since succession priorities are handed out in a randomized way, the results will vary between runs – two such examples are illustrated.

It is clear from the differences in the final tallies that the partially-randomized assignment of succession priority numbers causes the responsibilities of the missing site to be distributed among those remaining. As with any randomized algorithm, the optimality of the results varies from one run to the next – in this case, the second attempt gives a slightly more even distribution.
Aside from the final result, there is also the question of how much effort is required to perform all these succession negotiations. Table 10.9 summarizes the number of messages sent by each remaining site as negotiations came underway. Messages were logged starting when Site 3's heartbeat was missed, and ending once each site was aware of all newly-appointed masters, backups, and PG members. This particular data set corresponds to the first attempt mentioned above; the numbers will vary depending on how many elements each remaining site assumes the Master role for.

The message logs from this test run also demonstrate one of Datura's methods of fault tolerance. Site 2, evidently due to the high latency of its network connection, “jumped the gun” and
declared itself the new master for one of the resizable block's `ManipulatorTarget` elements (refer back to section 8.3.2, where this possibility was introduced). Site 5, which had the highest priority for that element but was slow to respond (at least from Site 2's perspective), sent out additional messages to ensure that every site was on the same track. This resulted in several more new master messages being sent out than would otherwise have been necessary.

Measuring the actual time required to complete these succession negotiations is problematic. None of the surviving sites know exactly when Site 3 disappeared; they only know when they detected the disappearance. With that in mind, table 10.10 explores each site's perceptions about the unavailable time of those elements that underwent a transfer. Figures are averaged over multiple test runs.

<table>
<thead>
<tr>
<th>Site 1</th>
<th>Site 2</th>
<th>Site 4</th>
<th>Site 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average unavailable (ms)</td>
<td>4208.4</td>
<td>4486.9</td>
<td>4310.3</td>
</tr>
<tr>
<td>Best case (ms)</td>
<td>2499.2</td>
<td>3581.6</td>
<td>2203.3</td>
</tr>
<tr>
<td>Worst case (ms)</td>
<td>6866.2</td>
<td>7025.3</td>
<td>6024.3</td>
</tr>
</tbody>
</table>

Table 10.10. Variation in unavailable times for elements undergoing succession negotiations.

The main factor in determining the unavailable time for an element is the time each mirror waits for a higher-priority mirror to declare itself the new master. There is a tradeoff between fast resolution and the possibility of errors such as Site 2's incorrect behavior mentioned above.

The ultimate goal of succession negotiations is to be left with a correctly functioning system. The correct state of the system was tested in the following ways:

- A combined role map of all sites in the CVE was generated; this was checked to ensure that each element had only one master mirror and one backup.
- The role map was checked to assure that the entities representing the Site 3 user's avatar were removed; this was confirmed visually at each site.
- The gravbots, whose masters were originally at Site 3, were visually observed at the remaining sites to ensure they were correctly synchronized.
• Several objects introduced by Site 3 were manipulated from other sites, to make sure their ManipulatorTarget elements were working correctly after the uncontrolled transfer.

Finally, a second set of tests was performed with Site 4 as the removed site. Results were similar to those described above, except that this effectively partitioned the network, with Site 5 cut off from the other sites. This “net split” situation is discussed more fully in the next section.

10.3.7. Test 7: Uncontrolled transfers and network splits

When we experimentally removed Site 4 from the network, a unique condition was created. The CVE was divided into two halves with no way of communicating between them. From the point of view of Site 3, for example, it appeared that both Site 4 and Site 5 had suddenly vanished, while from Site 5’s perspective, every other participating site had abruptly broken off communications. Table 10.11 shows the distribution of roles after all sites had completed succession negotiations.

The key thing to note is that elements can (and do) now have two masters – one on each part of the network. So long as these sites are separated, each side works independently. Complications arise should they ever be rejoined – if Site 4 is started up again, and reconnects to both sides.

<table>
<thead>
<tr>
<th></th>
<th>Site 1</th>
<th>Site 2</th>
<th>Site 3</th>
<th>Site 4</th>
<th>Site 5</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Before split</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Master</td>
<td>8</td>
<td>13</td>
<td>15</td>
<td>16</td>
<td>32</td>
</tr>
<tr>
<td>PG Member</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Backup</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Slave</td>
<td>74</td>
<td>70</td>
<td>68</td>
<td>68</td>
<td>52</td>
</tr>
<tr>
<td><strong>After split</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Master</td>
<td>24</td>
<td>22</td>
<td>23</td>
<td>(not present)</td>
<td>55</td>
</tr>
<tr>
<td>PG Member</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Backup</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Slave</td>
<td>44</td>
<td>45</td>
<td>46</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 10.11. Distribution of roles before and after net split. When Site 4 is removed, Site 5 becomes isolated and must reconstruct masters for its elements independent of the other sites.
This proved the most difficult situation for the sample implementation to deal with, and the
current solution, while generally functional, is less than ideal. The implementation takes a reactive
approach to detecting a reconnection following a network split. For example, suppose that mirror A
and mirror B of a particular element both are masters because of a net split. A sends a message, and
B realizes that only a master should be sending out that particular message. B sends a message re-
announcing itself. When A receives that message, it compares B’s succession priority (contained in
the announcement) with its own. Then A either abdicates (dropping to Slave mode) or sends out a
contradictory announcement (which would cause B, in turn, to abdicate).

This excerpt from one Site 5’s data log illustrates the process:

```
MShuey:ManipulatorTarget accepted
Usermanipulator:BasicManipulator
MShuey:ManipulatorTarget: send NEW_MASTER
(after receiving Site 1’s update)
MShuey:ManipulatorTarget: received NEW_MASTER (from Site 1)
taking over master from usurper
our pri: 0:2a: theirs: 0:4a:
MShuey:ManipulatorTarget: role MASTER
MShuey:ManipulatorTarget: send NEW_MASTER
(final message assures that all other sites get correct information)
```

The excerpt also demonstrates some of the “cross-talk” that can occur. In this case, both of the
masters of the ManipulatorTarget detected each other at the same time, since both responded
simultaneously to a request from a third site. Each site sent out a response to the manipulation
request, received the other’s response (triggering it to re-announce itself as master), and then
responded to the other’s announcement. The site whose output we’ve excerpted was the final master
due to its better succession priority.

While this results in only one master mirror existing after everything is resolved, the approach
has certain disadvantages. Since it depends on one master detecting a message indicating the
presence of another, discrepancies in the state of entities between those sites will persist until such a message happens. Furthermore, it is entirely likely that such a message would occur as part of a user interaction with the entity, and the flurry of renegotiation that it triggers could interfere with the user's interaction. Refinements to make this negotiation more proactive (without flooding the network with seldom-needed error checking messages) would be a valuable future improvement to the sample implementation.

10.3.8. Test 8: Uncontrolled transfers and user interaction

In the transfer tests considered so far, the application has been in a mostly quiescent state. No user interactions were occurring during transfers, and the pathfinder robots were inactive. The next several tests consider how elements respond to transfers that happen during activity. We begin by considering how uncontrolled transfers can affect user interaction and manipulation.

In the Test 8 scenario, a user located at Site 3 would manipulate (grab and move around) a resizeable block introduced by Site 5. The master mirror of the user's ManipulatorElement was located at Site 3 and was nontransferable — should Site 3 leave the CVE, the remaining sites would delete their mirrors of the manipulator rather than appoint a new master. The block's ManipulatorTarget's master mirror was located initially at Site 5.

During the test runs, one or the other of these sites would be removed from the CVE while the user was manipulating the block. The heartbeat signal (or lack thereof) from the missing site was used to detect the loss, and the system was monitored while it attempted to recover. The tests were performed multiple times to assure consistent results.

After each test, several checks were made to assure that the affected elements recovered to a sane state. The block's position was checked at each site for consistency. If Site 3 remained in the CVE, the user there would manipulate other objects in the environment to assure that his manipulator element was still working correctly. Also, users at several sites would attempt to manipulate objects whose masters had been at Site 5, including the block the Site 3 user manipulated, in order to verify
that the ManipulatorTargets and manipulation handler elements were responding correctly after succession negotiations.

The results can be divided into several specific cases, based on which site was removed and when this happened:

- Site 5 is removed before manipulation begins, and the user tries to grab the block before Site 3 detects the missing heartbeat. In this case, Site 3's mirror of the ManipulatorTarget sent out a request, but there was no master mirror to respond. When the request timed out, the Site 3 mirror rejected the manipulation attempt. Following succession negotiations, users (at Site 3 or elsewhere) could successfully grab the block. The ManipulatorTarget also sent a "missing master" message to its fellow mirrors, jumpstarting the succession negotiation process.

- Site 5 is removed before manipulation begins, and the user tries to grab the block after Site 3 detects the missing heartbeat, but before a new master has been appointed. Unlike the previous case, Site 3's mirror of the ManipulatorTarget knew there was no available master for it, and rejected the manipulation attempt immediately – it did not wait for a message timeout.

- Site 5 is removed while the user at Site 3 is moving the block, and the user does not release the block until after succession negotiations are completed. In this case, the disruption was completely invisible to the user at Site 3. The block continued responding to his movements throughout the succession negotiation process, and was dropped immediately when he tried to release it. Other users were then able to grab the same block.

- Site 5 is removed while the user at Site 3 is moving the block, and the user releases the block before the removal is detected by Site 3. During the initial run
of tests, problems were noticed during this test case. As expected, the block continued to follow the user's movements after he had attempted to release it (since there was no master to inform all the mirrors of the block to stop following). The block stopped moving when the new ManipulatorTarget master was appointed (and it had exchanged a verification message with the manipulator), again as expected. Subsequently, other users were again able to grab and move the block.

The problem was that when the block was eventually dropped it suddenly jumped to a different location in the virtual environment. After additional analysis, it was determined that the verification messages (which the newly-appointed ManipulatorTarget master used to determine if it should still be reacting to the user) were not sending adequate information about the user's movements. This was corrected, and while the block still followed the user's hand movements after being "released", once it stopped moving it stopped in place, without jumping to any strange location.

- Site 5 is removed while the user at Site 3 is moving the block, and the user releases the block while succession negotiations are underway. The results for this situation were identical to the preceding case, including the "jump on release" glitch.

- Site 3 is removed while the user there is manipulating the block. In this case, the block stopped moving as soon as Site 3 was removed (since no more position

---

21 This may seem like a poor way to handle the situation. The alternative is for the ManipulatorTarget to stop moving the block locally, while to users at remote sites it still appears to be moving. When the new ManipulatorTarget master actually sends out an update to those remote sites, the block would appear to "jump" back to where the Site 1 user thought it was released. Instead, the sample implementation chose the option in which all users see approximately the same results at all times.
updates from that user were received). The block's ManipulatorTarget still considered itself to be grabbed, and would not accept other manipulation requests, until the manipulator's mirror was removed at Site 5. This, in turn, happened when Site 5 detected that Site 3 had stopped sending heartbeat messages. When the manipulator was removed at Site 5, the ManipulatorTarget released itself, and users at the remaining sites were once again able to interact with that block.

- Site 3 and Site 5 are both removed while a manipulation is ongoing. In this case it was left for the remaining sites in the test network to sort themselves out. This test also revealed a defect in the sample implementation: ManipulatorTarget slave mirrors would crash if the local mirror of a manipulator was removed, and they were then promoted to the Master role. If these events happened in reverse order, the transfer happened correctly: the ManipulatorTarget master would detect the removal of the former manipulator, consider itself released, and be open for manipulation requests from other sites.

    The problem was traced to an incorrect ordering of checks in the ManipulatorTarget's update method; with this corrected both orderings of events produced the same end result.

With the correction of the minor flaws detected during these tests, manipulators and targets were able to survive the removal of any site even in the midst of an ongoing interaction. This is a key demonstration of Datura's goals of robustness in the face of unreliable systems and networks.

10.3.9. Test 9: Uncontrolled transfers and autonomous entities

As a further test of Datura's ability to transfer roles for elements even when those elements are actively engaged with the environment, a series of tests were conducted with uncontrolled transfers of the elements controlling the pathfinder robots in the test application. These tests had the dual goals of
demonstrating correctness for the three different pathfinder robots (and their different distribution patterns), and of showing the value of creating a backup mirror to improve performance and smooth over role transitions.

As a reminder, there are three versions of the pathfinder, each with a different element implementation controlling its behavior, AI and pathfinding routines: MSPathfinderElement (using the Master/Slave distribution pattern), MBSPathfinderElement (Master/Slave plus a backup mirror), and MPGSPathfinderElement (Master/Processing Group/Slave distribution pattern). For testing these elements, each executed its cargo pickup and deliver routine multiple times. The site hosting the master mirror of the PathfinderElement was removed at various points in this routine, and the robot was observed to determine how, or if, it would recover. Four points were tested:

1. While the robot is searching for a path to the cargo.
2. While the robot is following a path to the cargo.
3. While the robots has the cargo and is searching for a path to the landing pad.
4. While the robots has the cargo and is following a path to the landing pad.

We first consider the question of correctness: verifying that the robots always recovered in a "sane" state such that future interactions with the user and environment would occur as expected. Another aspect of this is considering how correctly the new master recreates the state of the robot before the uncontrolled transfer. For example, if the robot was actively searching for a path when the master mirror was lost, will the new master continue the search or will it fall back to a safe mode?

Correctness of the Master/Slave pathfinder. The M/S version of the robot was considered the simplest and tested first. In the initial version, very little status information was passed from the master to its slaves; the only time update messages were sent out was when the robot was following a path to either the cargo or the landing pad. This made it difficult for a new master of the controlling PathfinderElement, after an uncontrolled transfer, to decide whether it should be attempting to carry out its mission or if it should simply be "turned off". Eventually, a pair of
activation/deactivation messages were added to the MPathfinderElement's communications strategy, sent from the master to all slaves when they robot was activated and when it completed (or failed) its mission.

With these activation messages in place, the robot continued to carry out its mission after being interrupted during any of the four points listed above. However, due to the lack of available state information in the newly-promoted master mirror, it did perform in a less than optimal manner. The new master could only choose to initialize its AI routine in one of two states: shutdown (if it had not received an activation message from its predecessor), or reset (if it had). In the reset state, it essentially had to begin from scratch, finding a new path from its current position to the cargo or landing pad and then distributing this path to its slaves. This increased the normal recovery time required for the robot to continue with its mission (the particular performance issues are discussed below).

Correctness of the Master/ProcessorGroup/Slave pathfinder. The MFGPathfinderElement was modified with activation/deactivation messages as per the Master/Slave version above, and because of this operated in a similar fashion. Indeed, the recovery algorithm was essentially identical to the M/S version, and only the relative performance differed.

There was a possible optimization that was considered but not attempted with this particular robot. It involved the cases (2 and 4 above) where the robot was interrupted while following a path. Because of the particular way pathfinding was distributed, one of the PG sites would happen to have the particular path that the master chose to follow. If that site was chosen as the new master by the succession negotiation process, and if that site had recognized the path the robot was following as the one it had generated, it could conceivably continue to use that path, without searching for a new one. If this optimization had been implemented, the robot's actions would most likely have resembled those of the third variant, described next.
Correctness of the Master/Backup/Slave pathfinder. The final AI control element, `MBSPathfinderElement`, took a very different approach to dealing with uncontrolled transfers. It appointed one mirror as its backup, and kept that site constantly advised about changes in its master's internal state.

This had several advantages. For example, when the master found a path, the complete path was sent to the backup (instead of sending the path to mirrors one step at a time like the other variants). When the backup became master, it could immediately instruct the other mirrors of itself to continue along the original path. No extra searching was required.

Furthermore, since the M/B/S version had a better understanding of what stage of the mission it was at, it could avoid a certain amount of redundant checking. `MBSPathfinderElement` knew if it had already found its way to the cargo entity, and whether it had attempted to grab it or not. The other versions started from the beginning, releasing the cargo (if it had been grabbed) and picking it up again after they had verified their position.

This variant's benefits did not apply at all times. If it was interrupted at point 1 or point 3, while searching for a path, the new master still had to carry out the pathfinding routine from the start. Essentially, it acted just like the other two variants in this case. While the original master could have backed up additional information as the pathfinder algorithm was performed, this would have required a significant amount of bandwidth and was not attempted.

During this test, a peculiar interaction was discovered: Since the masters of both the `MBSPathfinderElement` and the cargo's `ManipulatorTarget` were initially located at Site 4, occasionally the robot would arrive at the cargo and be unable to immediately grab it. This would happen because the `ManipulatorTarget` had not yet finished its own succession negotiations (which we described in Test 8). This could theoretically happen with the other robot variants, but was unlikely because of the timing involved. Note that the possibility of such a result was anticipated in Chapter 8.
Figure 10.12. Recovery times for pathfinder robots when interrupted while following a path. This consists of two steps: the unavailable time during succession negotiations, and a second pathfinding step as the new master tries to orient itself.

<table>
<thead>
<tr>
<th></th>
<th>Unavailable time (ms)</th>
<th>Recovery pathfinding time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Master/Slave</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Average</td>
<td>4809.4</td>
<td>4881.4</td>
</tr>
<tr>
<td>Best case</td>
<td>4008.9</td>
<td>2452.0</td>
</tr>
<tr>
<td>Worst case</td>
<td>6011.6</td>
<td>8319.6</td>
</tr>
<tr>
<td>Master/PG/Slave</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Average</td>
<td>3016.1</td>
<td>4896.4</td>
</tr>
<tr>
<td>Best case</td>
<td>3002.5</td>
<td>2809.2</td>
</tr>
<tr>
<td>Worst case</td>
<td>3028.9</td>
<td>9718.2</td>
</tr>
<tr>
<td>Master/Backup/Slave</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Average</td>
<td>1009.6</td>
<td>0.0</td>
</tr>
<tr>
<td>Best case</td>
<td>1002.1</td>
<td>0.0</td>
</tr>
<tr>
<td>Worst case</td>
<td>1014.5</td>
<td>0.0</td>
</tr>
</tbody>
</table>

Performance comparisons of pathfinder variants. In order to quantify the performance difference in how the three pathfinder robots reacted to uncontrolled transfers, a simple test was performed. Each robot was interrupted at point 2 (following a path) during repeated runs. We took two measurements: first, the unavailable time as perceived by the new master after the interruption; and second, the extra time required by those variants that were forced to perform a new pathfinding check. These comparisons are summarized in the Table 10.12.

There are many interesting things about this table. First, consider the unavailable times. The extremely low values in the M/B/S version are to be expected: the backup site is certain that it can take over and can bypass most of the usual succession negotiation process. It waits only long enough to verify that the previous master is no longer responding.

The variation between the M/S and M/PG/S variants is more puzzling, as they should use the same methods for succession negotiation. This was eventually traced to a defect in the particular implementation of the MPGPathfinderElement: the formula for deciding how long a mirror should wait before taking over had undergone several adjustments during development, and one of these had not been applied to this particular class. This would have made "usurpation" - the wrong mirror announcing itself - more likely, though this did not actually happen during the test runs.
In the pathfinding section, the obvious difference is that the M/B/S variant skips this step completely (at least when interrupted at this point). The other two variants get much closer scores than would be expected based on their performance in Test 4, but this can be explained by looking at the overall state of the CVE and its sites. Recall from our network map that when we remove Site 4, Site 5 is also disconnected from the remaining sites. One of the three remaining sites becomes the \texttt{MPGSPathfinderElement}'s new master, while the other two become PG members. Usually, the PG group will consist of one of the Linux computers (Site 1 or Site 2) and the much slower Irix machine (Site 3). This eliminates most of the PG group's processing advantage over using a single site. We note also that the best case time reported for the \texttt{MPGSPathfinderElement} in the extra pathfinding step happened when Site 3 was appointed the master, and the two faster sites became its processing group.

At any rate, the results from this test clearly demonstrate a performance advantage for the M/B/S distribution pattern when an actively processing element is interrupted by an uncontrolled role transfer.

\textbf{10.3.10 Test 10: Transfers and processing group members}

The tests involving role transfers have concentrated on transfers of the Master role for various sites, but other roles can also be transferred. In Test 10, we examine the effects of transferring the PG member role for a mirror of an M/PG/S element.

For this test, we used the M/PG/S version of the pathfinder robot's \texttt{PathfinderElement} as the subject, and Site 1 as the site that would be forced to transfer the PG member role. This was engineered by setting the high load threshold at Site 1 to 10 milliseconds. As we can see from the load measurements in Test 3, this puts Site 1 over its high load threshold when the pathfinder begins searching for a path. The Location Manager at Site 1 then tries to lighten the load, and because of the high expected savings the \texttt{MPGSPathfinderElement} is the first pick of the selection policy. The only viable destination for this role is Site 5, since the other sites are all Masters or PG members.
already. This created a consistent situation for monitoring repeated tests.

As with the previous test, we are interested in both correctness and performance impact of this transfer. Correctness was judged by monitoring the output logs for Sites 1 and 5. Several particular actions were checked for:

1. The MPGSPathfinderElement mirror at Site 1 was always the first transfer candidate chosen by its LM. The LM implementation picks candidates in groups of three, but this was always first in line.

2. A bid was always received from Site 5 — and only from Site 5, though these were intermixed with bids for the "extra" candidates chosen by Site 1's LM. The PG member role was given up by Site 1 and assumed by Site 5.

3. Site 5, upon receiving the transferred PG member, announced itself correctly to the master mirror at Site 4. Since the pathfinding operation that had triggered Site 1's overload was still ongoing, the output logs were checked to ensure that the master confirmed the search request that Site 5 was then supposed to carry out, and that Site 5 began computing for that search request.

4. Finally, each test run ended with the pathfinding operation completing and the robot following the path chosen by one of the PG members. Subsequent pathfinding (e.g. from the cargo to the landing pad, and later activations of the robot) happened with all three PG members involved, and no additional transfers occurring.

During the preliminary test, an error was detected with the indexing of search requests sent out from the master mirror. This was quickly corrected, and the correct behavior was observed during ten subsequent runs of the pathfinder.

The performance question, in this test, was to measure the impact of the controlled transfer on the time required to find a path. The initial hypothesis was that it should have a small, but perhaps
measurable, effect. Remember that two PG member sites (Site 2 and Site 3) remained active while the third was being transferred from Site 1 to Site 5. Also, the controlled transfer process would begin very quickly once the mirror at Site 1 began its path search, so that relatively little computation was lost.

In fact, the average search time measured by this test was 7470 ms, slightly less than the average time measured during the original testing of the M/PG/S version of the pathfinder (in Test 3). While this test did not match the best time performance of the uninterrupted pathfinder (missing it by 1409 ms), it is clear that the transfer did not have a major effect.

10.4. Testing and evaluation summary

Taken together, these tests cover the most important and intriguing elements of Datura's LoC controls, and thoroughly test the ability of various elements to function correctly even while undergoing controlled and uncontrolled transfers.

The bulk of the test results were positive. Elements continued to behave correctly following various sorts of transfers, and the Location Managers made selection and location decisions that were in keeping with the design of their policies.

In several cases, defects in the implementation were located and corrected. In testing the ability of the pathfinder robots to survive uncontrolled transfers, deficiencies in the sharing of state were detected. These were resolved by a relatively small change – the addition of explicit activation and deactivation notifications for the slave and PG member mirrors. The most serious concern raised during testing is the behavior of the system as a whole when the group of sites are partitioned by a lost connection and later rejoined. While the tested elements were largely able to reintegrate themselves into a single, consistent group of mirrors, this was likely to cause noticeable user disruption. More aggressive management of such situations is a topic for future development, as we shall see in the final chapter.
11. Conclusion

11.1. Value and contributions of this research

This work has explored the technical requirements of collaborative virtual environments based on a number of application areas, and explored how existing CVE projects have addressed those requirements. This analysis was used to identify areas that could benefit from additional research.

The project undertaken here encompasses several aspects. First, of course, is the creation of a CVE library and several demonstrative applications. As with most such libraries, it provides a way to connect multiple sites into a common virtual environment, provides representations of users, and distributes knowledge about the objects that can be found in the environment. In common with several of the more advanced libraries, it allows for control and synchronization of user interaction with those objects, and allows new sites to add to the environment they join.

This basis was used to explore novel research in controlling and dynamically changing the location of computations for the elements that make up the virtual world. In common with recent CVE efforts such as Deva and Continuum, this work used elements (or "behaviors", "components", etc.) that could be combined to represent particular objects in the environment. In Datura, these became the unit of granularity at which LoC control was managed.

Whereas earlier efforts had allowed control to move between sites only under very specific circumstances, or only among a particular "virtual server" of sites with fast connections, Datura treated all sites as a peer-to-peer network. No site had particular explicit privileges over any other; instead, they are evaluated by each other only on basis of available resources and network connectivity. Datura allowed control to move from any site to any other site, at any time.

Datura also introduces a flexible Location Manager, which exists at each site and can be adapted to make LoC decisions in a variety of ways. In particular, the Location Manager in Datura
makes decisions to improve resource usage among participating sites, to improve the user experience (by making decisions that will minimally impact users), and to ensure the robustness of the CVE and its ability to survive network disruptions or failures at individual sites.

To meet these goals, the notion of computations performed by elements was expanded beyond the straightforward idea of placing a particular site in charge of a particular element. For example, elements have been designed so that many computations are performed locally to improve user perceptions of latency, with a control site only acting in a supervisory fashion to maintain synchronization. For computationally-intense tasks, sets of sites with available resources can be assigned to perform a task (via the M/PG/S distribution pattern). To allow smooth recoveries of elements with significant amounts of state information, other sites can be assigned to serve as backups, ready to take over should anything happen to the original.

It is hoped that these new capabilities will inspire additional research in the creation of reliable peer to peer collaborative systems, and that Datura itself will find practical usage outside the lab. It is particularly well-suited to handling small groups of dynamic sites, with various users coming and going over the course of a session, and its default element library provides many of the features needed for tasks such as collaborative walkthroughs and design reviews. Indeed, it is projected that the current Datura implementation will see real-world use at a number of facilities in the coming months.

We close with a consideration of some of the unanswered problems that have resulted from this research and evaluation, and with predictions about where development and research based on Datura's concepts could lead in the future.

11.2. Problems encountered

No research project of this scope proceeds without difficulties. Obviously, the design of Datura and the implementation of the library and applications was a long process of refactoring
designs, discovering bugs, and testing various approaches. Here we step back, and consider the handful of problems that still await their complete solution.

**Reliability in network layer.** Throughout the development and testing process, a number of problems arose in the underlying networking library. This library was itself in an early stage of development, but only receiving sporadic attention. On several occasions, work on the Datura sample implementation had to be suspended to explore the networking library's issues, and certain problems handling high-latency connections continued into the evaluation process.

**Difficulty of managing network splits and joins.** As discussed in section 10.3.7, the corrections that take place when a CVE is partitioned and then reconnected are not optimal. It can take an arbitrarily long time for a particular element to resynchronize all its mirrors, and this process is likely to be triggered by user interaction.

**Complexity of creating new elements.** The process of creating a new kind of element is more complex than we would desire. Implementing new element classes is an important part of creating unique and highly-interactive CVEs, and therefore must be accessible to most developers. As the sample implementation has evolved, a series of components have been created which a) hide much of the complexity of implementing the various distribution patterns; or b) provide detailed examples for the creation of new elements. However, enough complexity remains that it could prove a barrier to widespread adoption of Datura (or any tool based on its concepts). This also increases the difficulty of creating new applications for testing and evaluation purposes.

### 11.3. Future work

The completion of this research is a milestone, but not necessarily an end. There are many directions that continued research based on this project could take. In addition to addressing the shortcomings mentioned above, new research could enhance Datura's existing abilities, add major new capabilities, or simply increase the scope of applications that can be created with it.
11.3.1 Enhancements

Greater optimization of timing-based activity. There are several places where Datura makes decisions based on the time between certain events. An obvious example is the frequency of heartbeat signals from each site, and the delay before other sites recognize the absence of a heartbeat. Another example occurs during uncontrolled transfers, when lower-priority mirrors calculate a period during which they will wait for higher-priority mirrors to take charge.

In the sample implementation, these times are calculated fairly conservatively. However, this means that (particularly in the case of uncontrolled transfers when a site vanishes) the unavailable times because of negotiation may be longer than absolutely necessary. With further evaluation and larger sample sets, it may be possible to fine-tune these calculations, finding the best balance between minimizing wait times and having to correct for overzealous actions of some mirrors.

Proactive user-based location control. The use of location control in the sample implementation remains mostly reactive. Chapter 9 suggests measures that could be taken to more actively locate important roles to benefit the user experience. Primarily, this involves moving the Master role of entities/elements a particular user is likely to interact with to that user's site, in anticipation of his attempts to interact with them.

Load-balancing with more resource measures. The load-balancing algorithm in the sample implementation concerns itself primarily with the update time required for each element to perform its computations, and the position of users relative to candidate elements. Other factors could be considered, such as potential future computation usage, memory usage, and the network traffic received or generated by a particular mirror.

When determining destinations for transferred roles, the implementation implicitly favors sites with low-latency connections, but could be enhanced to more thoroughly consider network bandwidth and estimated transfer times.
**Resource scaling in heterogeneous CVEs.** It is common for CVEs to be heterogeneous, with different computing environments and capabilities at different sites. This makes it difficult to compare resource usage between sites, which in turn makes it difficult for a site to decide whether or not to bid for a given element's role during controlled transfers. The Datura implementation provided a simple scaling factor that could improve this situation somewhat, but this had to be manually configured at most sites. It would be better to provide an automatic way of determining how to scale resource usage measurements between sites, perhaps by creating a calibration test of some sort.

### 11.3.2. New major functionality

When we defined the scope of this project in Chapter One, several features of CVEs were deferred for future efforts and evaluation. We briefly reiterate that list here.

**Area of Interest (AoI) management.** In the Datura model as defined herein, all sites contain mirrors of all entities and elements; when an element updates its state it updates it everywhere. This presents a bottleneck as the number of users and omnipresent entities in a CVE increases. An AoI capability in Datura would reduce the bottleneck by only requiring a site to be aware of a subset of entities – typically those that could be perceived from the user's location in the virtual world.

**Security.** In the sample implementation, Datura sites trust information coming from (or claiming to come from) all other sites. But a framework exists that could allow for enhanced security capabilities. The sites that are viable transfer targets for a particular element can be controlled or defined by the current or primordial master mirror of that element, simply by deciding whether or not to give a particular site a succession priority number. These priorities could themselves be signed to prevent forgeries. Similarly, an authorization check could be performed when a user attempts to interact with a particular entity.

Encrypting the communications channels between sites is another important aspect of securing a Datura environment; this would prevent eavesdropping of important information, as well as injection of false data or commands into the communication stream.
Integrated voice communication. When Datura has been tested over large geographic distances, voice communication has been done with traditional conference telephone technology. This requires a separate communications system to be set up, making it more difficult and more time consuming to join a Datura session. Transmitting voice communications of users over the same Plexus network used by Datura could simplify this setup procedure, so long as the connecting sites have adequate bandwidth. Voice communications could also be integrated with AoI management to allow finer controller over which users are communicating with each other.

11.3.3. Scope of available and future applications

The ultimate motivator of this research is that Datura, or a successor built upon its best ideas, should receive widespread use in the real world. As more applications are written using these concepts, the library of available elements will be improved and expanded upon, as will our understanding of the role of location in the functioning of CVEs. This could lead to improved transfer techniques, or new distribution patterns, or more holistic location management algorithms. These could prove invaluable as the interest in using CVEs increases in coming years.
Bibliography


[Burdea/Coifet94b] Ibid., pp. 308-310.


[id96] id Software, Quake. 1996.


(Retrieved July 2004).


[Murthy/Manimaran01] C. Siva Ram Murthy and Govindarasu Manimaran, Resource Management in

[Murthy/Manimaran01b] Ibid. pp. 143.

[Murthy/Manimaran01b] Ibid. pp. 141.


July 2004).

July 2004).

[Ogi+Ol] Tetsuro Ogi, Toshio Yamada, Ken Tamagawa, Makoto Kano, and Michitaka Hirose,
"Immersive Telecommunications Using Stereo Video Avatar". Proc. IEEE VR 2001, Yokohama,


Environments (and Online Games)". ACM SIGGRAPH 2001 Conference Abstracts and


University. 2002.


