Continuum:

An architecture for user evolvable collaborative virtual environments

by

Allen Bierbaum

A dissertation submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Major: Computer Engineering

Program of Study Committee:
Carolina Cruz-Neira, Major Professor
Doug Jacobson
Jim Davis
Chris Harding
Diane Rover

Iowa State University

Ames, Iowa

2007

UMI Number: 3289395

ProQuest Information and Learning Company
300 North Zeeb Road
P.O. Box 1346
Ann Arbor, MI 48106-1346

TABLE OF CONTENTS

LIST OF FIGURES

ABSTRACT

Continuum is a software platform for collaborative virtual environments. Continuum's architecture supplies a world model and defines how to combine object state, behavior code, and resource data into this single shared structure. The system frees distributed users from the constraints of monolithic centralized virtual world architectures and instead allows individual users to extend and evolve the virtual world by creating and controlling their own individual pieces of the larger world model. The architecture provides support for data distribution, code management, resource management, and rapid deployment through standardized viewers. This work not only provides this architecture, but it includes a proven implementation and the associated development tools to allow for creation of these worlds.

1 Introduction

1.1 Motivation

Futurist and science-fiction writers have long envisioned shared virtual worlds — places where millions of people enter a shared alternate reality with no physical matter or constraints. Participants would be able to meet, socialize, and work in a completely virtual world that only exists in the collaborating minds of the computers controlling it. Many of the more idealistic visionaries look to this virtual world as a cyber utopia; a place without borders, without strife, without limits. This new cyber world could be shaped as the inhabitants see fit with the only limits being the imagination of its creators.

These ideas have inspired and intrigued computer researchers, developers, hackers, and users since they first started to emerge. But the goal of shared virtual worlds, while a tantalizing dream, has always seemed to be unattainable. Much like a mirage, always just out of reach but so very enticing.

Fortunately with recent technological advances, including plentiful processing power and abundant network resources, this mirage has started to come into clearer focus. We have arrived at a point in time where the pursuit of shared virtual worlds is not an impossible dream, but instead is a feasible reality.

There are currently several efforts working on systems to allow groups of remote users to share a virtual world. These systems are called collaborative virtual environments (CVEs). CVEs are a combination of hardware and software that present users with a computer generated representation of a shared virtual world. A virtual world provides the description of the physical and behavioral characteristics of the space being presented. It may define any imaginable environment from a simulated manufacturing process to a scientific visualization

or the latest game.

CVEs enable multiple users to socialize and interact within a shared virtual space. They give all users a sense of shared *presence*; a sense of being "in" an application and feeling that they are all together in the same space. Users are not just passive observers or indirect controllers, instead applications are designed to engage users and provide them with a sense of personal involvement as if they are part of the virtual space.

CVE applications open up a wide range of possibilities for many application domains. Users from around the world can come together to collaborate in a shared space gaining new insight into problems, evaluating new products, and even engaging in entertainment. CVEs have been used for scientific visualization, manufacturing, entertainment, and nearly every other domain where standard VE technology is currently being applied [Eve, Ult, Lab, Har01, SZ99].

Consider an example where an international engineering company wants to use a CVE to review a new product design. Engineers from offices in Chicago, London, and Tokyo need to discuss the design the same way they would if they were all meeting together in the same room with a physical prototype of the design. But instead of using a physical object they are using a virtual environment and instead of being in the same room they are all collaborating in a CVE. The CVE allows engineers from every location to be simultaneously in the environment reviewing the virtual design without having to spend the time or money to build a physical object and travel to the same location.

CVE systems are usually implemented using real-time interactive software applications that produce a fully synthetic computer generated representation of a shared virtual world. Normally this representation is multi-sensory including three-dimensional graphics and high-fidelity sounds. The software is connected over a network to a set of remote user applications. It is responsible for exchanging data with all the distributed users to support collaborative interaction and maintain a consistent view of the shared virtual world. Because of their similarity to standard virtual environment (VE) applications [1] many CVE systems build upon

[1] In the context of this document, we refer to VE software systems as presenting only a local virtual world. They do not have support for sharing the environment with other users.

existing VE architectures.

Current implementations can effectively support hundreds and sometimes thousands of collaborating users. Going forward though we see the need to support much larger CVEs where there may be hundreds of thousands of users. These environments are called large scale collaborative virtual environments (LSCVEs) and have an additional set of requirement due to their immense complexity.

The leading adopters of LSCVE technology are currently the game development community. They have been successfully using LSCVE technology for several years to create massively multi-player on-line games (MMPOGs) that allow thousands of players to inhabit and interact within the same gaming world [Ult, Lin, Eve, Lab]. The popularity of these games has led many game companies to start developing games with collaborative features. This wide-spread development and deployment has thus become a major driving force of current LSCVE research.

Although CVEs are currently being used with success, there are still areas that need to be improved upon. One such area is virtual world extension. Most existing CVE applications are constrained to one specific predefined application and associated virtual world. These systems do not allow individual users to extend the virtual world by adding new content, instead extension is restricted to a single centralized authority that controls the entire virtual world.

A more intriguing view of the future of CVEs is one where the end users actively contribute and control content in the shared virtual worlds. Users could add new geometric information about how some item looks in the environment or add new code for controlling a shared object or a section of the environment. But how can this type of shared control be allowed in a CVE? This research sets out to address this question by examining the fundamental questions of virtual world extensibility.

Evolvable extension We believe that to allow the extension of virtual worlds, CVEs need to support *evolvable extension*. CVEs supporting evolvable extension allow users to add to and extend the environment. The virtual world is not restricted to only what the application

vendor originally put in, but evolves to the requirements and directions of the users.

As an example, consider a virtual city within a CVE. Like most real world cities this virtual city has a central district with areas for socializing with friends and meeting new people. This district is full of entertainment attractions such as movie theaters, music clubs, cafes, and general outdoor meeting areas. There are also stores and vendors selling their latest wares ranging from the latest real-world fashion or music to virtual accessories that can only be used within the virtual world.

In the menagerie of users and uses that make up such an environment, it is actually the users of the environment that determine how the virtual world is structured and how it functions. The users need the ability to add to the environment itself and extend it as needed. Whether it is shop owners refining the virtual representation of their shops to attract more customers, a street performer tweaking the flaming torches she is juggling to make them look and behave just a bit more dangerous, or city officials creating a central park for strolling. In each case, the environment as a whole is composed not of one single application but many separate "mini-applications" and environments that, when put together, form the actual city center.

If this environment sounds similar to environments portrayed in popular science fiction such as the Metaverse[Ste92] or the Matrix[WW99], it is not just by chance. Although these fictional environments represent a much more far reaching culmination of LSCVEs, they are related to modern systems. It is doubtful that the progeny of today's LSCVEs will be exactly what science fiction currently portrays, but the end goal possesses many of the same features.

We do not know how the evolution of LSCVEs will end, we can only explore how it will begin. For now we will pull back the dream world and start addressing the challenges of the real world. By addressing the technical challenges of today's LSCVEs we can begin taking the first steps toward a future where they are a reality instead of a work of fiction.

1.2 Research Goal

The goal of this research is to design a software architecture for supporting LSCVEs where all users can extend and control the shared virtual world.

The time is right for creating architectures for LSCVEs. Only recently have the required technologies reached a level of maturity and capability that allows the creation of such a system to be more then a fanciful endeavor. The abilities of networking, graphics cards, and CPU processing power have aligned themselves in such a way to make such systems feasible. Combine these hardware capabilities with the advent of software technologies such as secure component-based software and peer-to-peer networking and it is clear that we have reached a new level of capabilities. We can now take the first steps toward world-wide collaborative virtual environments. The question now is not when will we have the capabilities needed, but is instead how will we use the capabilities that exist to make these systems a reality.

1.3 Challenges

Creating CVE software systems presents a number of difficult challenges. Many of these challenges are a consequence of the intrinsic complexity and scale of allowing for a large number of users simultaneously accessing and sharing the same virtual world. These users have a wide range of hardware with vastly different computational power, networking resources, and interaction capabilities. In addition to hardware issues, there are the software issues inherent in developing complex distributed algorithms. These algorithms operate over a network and must handle resource constraints, communication failures, and unexpected disconnections.

There are also social and behavioral challenges that need to be addressed. These include usability issues such as how users collaboratively interact with each other and how they find their way through the virtual worlds. There are also issues related to why a person wants to be in a CVE and how they behave in a virtual world.

Although these are all important issues that must be addressed, this research only focuses

on the software issues and more specifically on the design of software architectures to support CVEs. These include issues such as how to represent the shared virtual world, support distributed interaction, scale to large numbers of users, share the state of the virtual world, and develop applications.

The challenges of CVE software have been addressed to the point that CVEs can be used today. However, there are still unsolved challenges that prevent CVEs from being used to their full potential.

1.3.1 Lack of a common model of the virtual world

There is little or no support for reusability or compatibility between CVEs. Because each CVE that is developed represents the world in a different way there is no common method for sharing object data and application code between CVEs. The CVE resources are instead restricted to the CVE for which they were developed.

1.3.2 Virtual worlds are not user extensible

Most current CVE systems rely upon each user running exactly the same application with centrally controlled code and content. This setup does not allow for virtual worlds that are composed of the collective contributions of the users. Instead, the users are restricted to a passive and static world that is not under their control. This can support basic collaboration, but it does not allow for the amount of user control that is needed to build virtual communities. To allow this degree of flexibility, the system needs to support full user extensibility — what we have called evolvable extension.

1.3.3 Deployment is difficult

Deploying CVEs is currently difficult and error-prone. Users have to download the application executable and all associated data files. If there is a change to the application or supporting data to fix a bug or add a feature, then the user has to download this update. This manual process leads to problems where users may not be running the most recent version of

the application or do not have the most recent version of the static data that the application uses. Because of the amount of manual interaction required the opportunity is very high for mistakes and incompatibilities that cause the client application to run unsuccessfully.

1.3.4 Application development is burdensome

It is currently a major undertaking to create a CVE application. Developers have to contend with all of the development issues described earlier and many times they do not have CVE-related tools to help them with development. Before CVEs can be widely developed and deployed system designers need to provide better tools for supporting application and content creation.

1.3.5 Limited number of users

Most current CVE architectures limit the number of simultaneous users. These limits exist because as the number of users increases the amount of supporting resources needed also increases until they reach the limits of the supporting architecture. Until scalable solutions exist that can adaptively balance the resource constraints in the system, CVEs will remain limited to a relatively small numbers of active shared users.

1.4 Problem Statement

The goal of this research is to develop a software architecture and an associated unified world model for large-scale collaborative virtual environments. The world model will combine code, behavior, and data into a single common structure that is user extensible. The contributions of this research are:

- The design of a unified world model for CVE applications

  The world model in this research is unique because it allows code, behavior, and data to be combined into a single structure. There have been previous attempts to do this, but each of these methods has limitations that need to be overcome.

- The ability for CVEs to be composed of and extended by user contributions

  The world model in this research supports user extensibility. Users provide and control the content and behavior of the virtual world.

The research methodology used by this project is one of exploration through implementation and iteration. We will verify our ideas by evaluating prototype implementations. The evaluation will use test-case applications designed to mirror real-world requirements.

Through the experimentation of this research there were several concrete deliverables. These include:

- Unified world model
- Network communication system
- Client viewer for participating in the CVE
- Common software tools and components for supporting application development

1.5 Research Issues

An architecture that addresses this research problem must unravel several interrelated research issues. In this section we provide a high-level overview of each of these research issues. In later sections we will describe in detail the methods used to address each of these issues within the scope of this work.

1.5.1 Unified world model

The primary research issue to be addressed is how to represent the virtual world for a CVE. This representation defines how applications view the virtual world structure and how state is shared between user applications. Most CVE systems implement a shared data structure for capturing this representation. We call this representation of a virtual world and its associated implementation the *world model*.

Previous efforts use a wide variety of methods to represent this structure. Several systems extend scene graph data structures with non-graphical application information [FS98, Mas]. A number of other systems use data structures that facilitate distributed shared memory or potentially distributed objects [Pet99, PCMW00]. Some systems are not meant to be used for in memory representation, but are only used to specify the structure of the virtual world [VRM97a, Liv03].

The world model proposed in this research extends the standard idea of a world model by representing a virtual world as a hierarchical composition of entities and related code components. The *entities* store the state of all objects in the virtual world. The associated code components control how the entities respond to interaction and behave within the virtual world. The hierarchical structure of the world model defines the authority and spatial relationships of the entities. This structure forms the backbone of the virtual world and provides a *unified world model* that allows application code from many different sources to inter-operate within a single environment.

Entities provide a common basis for capturing shared data in the virtual world. Each entity represents a physical object or space in the virtual world by holding a set of properties that fully define the state of the object they are representing. These properties may be intrinsic characteristics of the entity (like the color of a ball) or they may be application-specific data that is added to store information needed for a custom algorithm. Additionally, each entity has associated metadata[2] that describes its current properties. This metadata allows run-time application code to interrogate an entity's properties, add new application specific properties, and discover new entity types all at run-time.

We will refer to the code components in the world model as *codelets*. The codelets can be thought of as mini-applications that control entities in the virtual world. For example there could be codelet attached to a fountain model that controls a water particle system. Codelet may also control sub-regions of a virtual world. For example a virtual room may have all of its physics controlled by a single codelet associated with its containing entity.

[2]Metadata is data that describes other data[FY98]. This is a machine readable description that allows application codes to operate on new data without having to know the exact layout of makeup of the data a priori.

A unique aspect of this architecture is that codelets in the world model can be contributed by any user of the environment. Since any user can contribute to the environment, it is effectively running many user "applications" simultaneously to create the shared environment. There are obvious security concerns in any system that allows user supplied code to be executed. In later sections we will discuss the security details of codelets.

1.5.2 Data distribution

Another research issue that must be addressed is how to design a data distribution system that effectively shares the current state of the shared virtual world among multiple users. As described in the previous section, the world model is composed of many entities which each hold a portion of the data representing the shared virtual world. Thus for the virtual world to be shared, each of these entities has to be distributed to all the users in the environment.

Distributing the world model data is a difficult problem to solve efficiently. CVE system developers must choose between updating the data with very low latency while consuming an enormous amount of networking resources or delaying the data updates at the expense of increased latency. At the same time the data consistency strategy must guarantee that all users in the shared virtual world share synchronized data values.

Even minor problems with the data distribution implementation can lead to many unwanted artifacts within the system. Faulty data distribution can lead to invalid or corrupt data. Even if the system delivers the correct data, if the data is not distributed quickly enough users will perceive lag within the system. In extreme cases this lag can render the system useless by preventing interaction. These types of problems are beyond the control of the application developer and must be solved by the CVE architecture designers in a reusable data distribution system. This system can then be used by application developers and will ensure correct data distribution behavior.

The data distribution system is responsible for:

- Holding entity state information
- Distributing entity state updates to all CVE users

- Controlling access to entities
- Synchronizing access to entities

Another aspect that is commonly managed by the data distribution method is data persistence. When the CVE is unloaded or some sub-component of the CVE is stopped, the state of each affected entity within the CVE needs to be stored so that it can be loaded later when needed.

1.5.3 Code management

Another research issue is how to manage the many codelets that are part of the virtual world. As described earlier, the world model associates application codelets directly with entities in the world model. This allows the world model to specify how the codelets should relate to the rest of the world, but it does not define how they should be managed. A separate management system is needed to handle the codelet execution, coordination, and security.

We propose that codelets can be effectively managed using a *component system*. Component systems allow applications to be created by combining modularized units of code (called *components*) into a larger application [SGM02]. In our case the software components correspond directly to codelets. The use of a component system helps to solve or at least assist with many of the basic code management issues. For example many component models [SGM02, Rog97, Box97, Pri99, Ham96, MS01, HV99, Obj02] support the concepts of execution, dependency management, building, versioning, and package signing. There are several extended features that are needed for a component system for CVEs.

One desirable feature is multi-language bindings. These allow developers to use multiple languages for developing and using components. For example we may want to have an application that implements the low-level graphics rendering in C++ but has the interaction code written in a high-level scripting language like Python. By allowing higher level scripting languages to be used we may be able to reduce the complexity of application development. An added benefit of multi-language bindings is that developers can use virtual machine based

languages like Java [AG98] and C# [ECM02] to allow direct migration of compiled applications across platforms with minimal effort.

Component security is also a requirement for a CVE component system. When developing applications that are composed of components downloaded from remote locations, security becomes a major concern. There is always a risk that downloaded components may contain malicious code designed to harm a local system. Some component systems have support for security, but we believe this is one area where additional extensions may need to be added to handle the requirements of CVEs.

An important and highly unique requirement for LSCVE is support for continuous execution. When administrators or users upgrade or extend a LSCVE it may not be possible to take down the entire system. Instead system components need to allow upgrade and maintenance while they are executing. This is not supported by any component system we currently know of, but is the topic of several active research efforts [Wat98, WZ98, OCS00, BZWM97]. Unfortunately there is no current component system that solves all the technical issues that we need for creating CVEs.

1.5.4 Resource management

A crucial research issue is how to manage the resources required by virtual worlds. The world models of LSCVEs depend upon many different resources ranging from geometric model files that specify how objects appear to code components that define how objects behave and interact. We use the generic term *resource* to refer to any static item or component that a world model requires. For example the world model from the virtual city example may require a model resource that defines how a building looks, a code resource that implements a fountain in the city park, and a sound resource that contains music being played in a club.

The resource management strategy is part of the underlying software used to support the world model. The responsibilities of this system include:

- Resource retrieval

When an application depends upon a resource, it needs a way to find and retrieve the resource. To support this functionality, resources need to be identified and indexed so they can be located. The system also needs to provide a method for retrieving the data from the remote nodes that have the requested resource.

- Version management

  Just as in real life, the virtual world does not remain static. It changes over time. Resources used by a CVE will most likely change over time. For example the model file describing the park in our virtual city example may be updated many times to add more detail or to change the feel of the environment. These changes need to be reflected in the resources to allow the system to request specific versions or just the most recent revision.

- Authentication

  When the resource manager retrieves a resource it needs a way to verify the resource is authentic. This includes verifying that the resource was created by the correct user and that it has not been altered from the original version.

1.5.5 Space sharing rules

One of the most challenging research issues addressed by this work is creating methods to resolve conflicts when applications overlap in space or behavior. Because the world model allows virtual worlds to be composed of contributions from multiple users there are effectively multiple applications running simultaneously inside the environment. In a CVE where multiple applications are running, there are going to be places where applications "overlap" in space. These overlaps lead to conflicts were the overlapping applications disagree about how that area of the virtual world should be presented and which codelets are in control. The CVE system needs to provide a method to resolve these conflicts in an orderly and fair way.

For example consider again the case of the virtual city. Imagine that within the city there is a building owner. The city administrators have granted the building owner the rights to

control an area of the city where they can put their own building and associated application code. The building owner specifies the layout of the space, the geometry used to represent it, and the interaction code that will execute within the space. But what happens if the owners specifies a model that is larger then the allocated space? Correspondingly, what if the owner defines a different gravity constant for the space inside the building (like zero gravity) but the city managers have specified that gravity should remain consistent across the entire city? In both of these cases there is a conflict between pieces of code sharing space within the virtual world.

Resolving these conflicts effectively must be done if we are to allow all users to extend and evolve the virtual world. Without conflict resolution there would be no way to prevent adversarial users from usurping control of areas that are rightfully under the control of other users. In effect if conflict resolution rules are not in place it becomes impossible to support shared control of a virtual world effectively.

1.5.6 Development tools

It is not enough to have a CVE architecture that addresses only the previously mentioned technical issues. If developers do not have the tools they need to create applications, the architecture will be of no use. Because of this we do not limit our focus to creating only a CVE architecture. Instead we believe that supporting developers with tools for creating and extending virtual worlds is another crucial research issue.

Unfortunately, CVE application development has traditionally been notoriously difficult. One reason for this is developers have to manage the complexities of all the technical issues described previously including real-time graphics, distributed algorithms, and human computer interaction. This complexity reduces developer productivity by requiring them to handle the low-level architectural details instead of focusing on the development task at hand: creating an application. Complexity further hinders development by making it difficult to find developers that possess all the skills used developing CVE application. For this reason fewer applications are developed and many that are developed do not take advantage of the

full potential of the CVE system.

We believe that it should be the responsibility of the CVE architecture designers to provide development tools that assist application developers in creating applications. As part of our CVE research we will analyze the needs of application developers and design development tools to help meet those needs. It is our hope that these tools will assist application developers in creating new applications and will lead to higher adoption rates for the CVE architecture produced.

1.6 Contributions and results

This research describes an architecture for a unified world model that supports user extension. The world model combines code, behavior, and data into a shared structure that is user extensible. The architecture provides support for data distribution, code management, resource management, and rapid deployment through standardized viewers. The system is based on a layer architecture composed of networking, data sharing, a world model, and viewers.

1.7 Dissertation Overview

This dissertation is organized as follows:

Chapter 2 provides an overview of the virtual environment, networking, and collaborative virtual environment technologies that the rest of the work builds upon. This section does not go into explicit detail about every nuance of these technologies, instead it is meant to provide a broad overview to give the reader a foundation for the discussions in the remaining sections of this document.

Chapter 3 reviews previous work in related areas. It starts out by describing the details of several previous architectures for CVEs and discusses the relationships between then. The section then discusses more peripherally related systems that have interesting aspect that relate to the issues addressed in this work.

Chapters 4-7 describes the architecture of Continuum and the various layers that make up the system.

Chapter 8 discusses the conclusions that can be made from this work and future work that could follow on this research.

2 Background

Collaborative Virtual Environment (CVE) software systems build upon many different fields including computer graphics, computer algorithms, software engineering, human computer interaction, simulation, and networking. These fields cover the technologies needed for defining the virtual world, simulating the elements within it, distributing the virtual world to all users, presenting it to each user, and facilitating user interaction.

In this chapter we present background material covering some of the details of these technologies. Because it is beyond the scope of this document to cover the details of all the technologies used for CVEs, we limit the background material to those aspects that are unique to CVEs. There are many excellent references that describe the technologies that CVEs have in common with other software systems including virtual environments[Stu96, Vin95], visual simulation[WW92, Fol90, Wat93, Bou02, Mol99], and game development [Ebe01, DeL00, DeL01, Tre02, Bou02]. We highly recommend examining these references for a more complete description of background technologies used in developing CVEs.

We begin this chapter with an overview of networking, we then describe the fundamental characteristics of a CVE, and we end the chapter with a detailed description of many of the issues involved in developing CVE systems.

2.1 Networking Overview

Because CVEs use a network to share information between users, many of the topics in this document involve the subject of networking. Throughout this document we will use the generic term *network* to refer to the variety of transmission media, hardware devices, and software components that make up a computer network. Strictly defined, a computer net-

work is an interconnected collection of autonomous computers [Tan96]. Our more generic definition of network encompasses the transmission media, including wire, cable, and wireless channels; the hardware devices, including routers, switches, and hubs; and the software components, including protocol stacks, communication handlers, and drivers. Computers or devices that use the network for communication are called *hosts*. Additionally, any connection or routing point within the network such as a computer or switch is called a *node*[CDK01].

To simplify our discussions, we will ignore the specific details of the networking hardware used (unless otherwise noted) and instead focus on a model of the network as a communication system. Within this model the network is a medium that allows hosts to communicate with each other by exchanging data in the form of messages. Although this model ignores some of the hardware level details such as switching and packet assembly, it provides a level of abstraction that is appropriate for our discussions.

There are several fundamental characteristics of network communication that affect distributed communication. The first of these that we will discuss is network performance.

2.1.1 Network performance

Network performance is a key issue for any distributed algorithm because they are constrained by the performance limitations of the network. Network performance limitations are such an important issue, that the complexity in many distributed algorithms is introduced primarily by techniques deal with these limits.

The primary performance characteristics of a network are latency, bandwidth, and jitter [CDK01].

- The delay from the sending of a message on one host to the receiving of the message by another host is called *latency*. Latency can be measured by sending an empty message between two hosts. Latency measurements include the following time measurements:
  - Network transmission time: This is the time taken for the first bit of data transmitted across the physical network to reach the destination.

  - Network access time: This is the delay introduced in accessing the physical network. This delay includes access time at both the sending and receiving hosts. An example of this is the time spent waiting for an Ethernet line to be available for communication. Because access time increases as more nodes try to access the same shared network resources, network access time can increase significantly during periods of high network traffic.
  - Processing time: This is the time taken by the operating system and associated communication software to process the message at both the sending and receiving end. This processing includes tasks such as adding headers to the data, segmenting data packets on the sending side, and applying the opposite behavior on the receiving end. The receiving end may also buffer the received data which can add further delay to the processing time. In some cases, the processing delay also includes application processing time. In such cases, we will explicitly state that this delay is being considered as part of latency.

- The network *bandwidth* is the aggregate amount of data that can be transmitted across a network connection during a given time period. Network bandwidth is shared by all communication channels that are using a given part of the network.

- *Jitter* is a measure of the variation of the transfer times of a sequence of message transfers. Jitter can either refer to the latency of the messages or to the total transfer time of the messages. Unless otherwise noted, we will use jitter to refer to the variation of the total transfer times.

The bandwidth and latency of a network channel determine the total time taken to transfer a piece of data between two hosts. Assuming constant bandwidth during a transmission, the total time to transfer a message of size *length* can be computed as:

$$Message\ \ transfer\ \ time = latency + \frac{length}{bandwidth}$$

From this definition it can be seen that any increase in latency or decrease in bandwidth can dramatically increase the time needed to complete a message transfer. It can also be seen

that no matter how little data is transferred or how high bandwidth the connection, there is at least *latency* delay in any message transfer. Because latency incorporates delays tied to physical transmission, there is no way to completely eliminate it. We will discuss this more later but it should be noted here that the inability to remove network latency is the key restriction for CVE communication.

2.1.2 Network reliability

In addition to performance limits, developers must also account for issues of network reliability. The reliability of network communication is not guaranteed. Network transmissions can fail due to many reasons. A physical connection may go down. A data buffer in the software system may overrun. Depending upon the network protocols being used, transient failures may be automatically corrected. If they are not automatically corrected, then application developers need to write their software in a way that detects the error and keeps functioning correctly. This may mean the software corrects the error through methods such as retransmission, or it may mean that the software just ignores the erroneous message. Alternatively, if the protocol being used guarantees error-free transfers, then the software developer only needs to worry about more permanent error conditions such as network disconnections and failures.

2.1.3 Communication architecture

The system architecture used by a set of distributed applications communicating over a network is called a communication architecture. The communication architecture defines the responsibilities of the individual communicating nodes in the system and determines how the application is distributed within the system. The choice of communication architecture used by a distributed system can dramatically impact the performance, scalability, reliability, and complexity of a given system. Most network-based applications use one of two types of communication architectures[1]: client-server or peer-to-peer.

[1]There many hybrid communication architectures that combine aspects of both of these methods, but for the purposes of this discussion we will only discuss these two types. For a more complete description of communi-

2.1.3.1 Client server architecture

The client-server architecture defines a communication relationship where a client application communicates with a remote server system. The server system provides access to services or resources that are either held locally on the server or are under the control of the server. The client system communicates with the server to request access to the server's services and resources. It is then the server's responsibility to fulfill the request and possibly respond to the client with the result of the requested action [CDK01].

A common example of a client-server based system is the relationship between a web browser and a web server. A web server manages the web pages for a site and is responsible for processing all webpage requests for that site. When a user wants to visit the site they use a web browser application which acts as a client. The web browser requests the web page from the remote web server. The server then replies with the content of the requested page. When the client receives this information it displays it for the user and waits for the user to request another page.

2.1.3.2 Peer-to-Peer architecture

A peer-to-peer (P2P) architecture is one where all the connected hosts have similar roles, communicating as *peers* with no distinction between client or server roles[CDK01]. Each peer contains all the algorithms necessary to function in any peer role. Each peer is an equal participant with no hosts providing specialized facilities or administration. Because there is no distinction between peers and no host has unique abilities that the others lack, P2P networks provide a fully decentralized system that can be very robust[Ora01].

There are many example of P2P networks in widespread use today. Some of the most well known are file-sharing networks like Gnutella [Gnu03] and Freenet [Fre03] but these are just a few of the systems in use. Usenet [SL98] and DNS [AL01] are examples of P2P systems that have been in use since the early 1980's. There are also numerous examples of hybrid systems that are primarily P2P such as instant messaging and Napster [Ora01, Ada01]. These

cations architectures see [CDK01, SZ99].

applications all benefit from a decentralized architecture that allows them to efficiently share information directly between hosts.

2.1.4 Communication protocols

Data is communicated over a network using communication protocols. There are many available communication protocols, but we will be focusing exclusively on the TCP/IP[2] suite of protocols[Ste98, Ste94, WS95]. We limit ourselves to discussing these protocols for two reasons. First because they form the basis of the Internet, they are currently the most widely used networking protocols. Second, they form the basis of the networking software used for this research.

2.1.4.1 Transport protocol

The primary means of communication between two hosts on a network is by sending messages from a sender to a receiver. Transmitting these messages is the responsibility of the network transport protocol being used. TCP/IP supports two transport protocols: TCP[3] and UDP[4]. These transport protocols are in turn based upon the IP network protocol. It is the responsibility of the IP to provide data delivery for the higher level transport protocols.

When an application process wants to send data to another host they must provide two pieces of information: the message data and a destination address. The message data can be any data that the process would like to communicate. The destination address is an identifier that tells the network protocols where to deliver the message data. The application does not have to explicitly identify themselves as the source of the communication. The protocols will automatically add the sender's identification into the source address.

In the case of the TCP and the UDP, source and destination addresses are composed of an IP address and a network port. An *IP address* is a unique numeric identifier assigned to each host on the network. This identifier is used by the IP to deliver data to the correct host on the

[2]TCP stands for Transmission Control Protocol, IP stands for Internet Protocol.
[3]Transmission Control Protocol
[4]User Datagram Protocol

network. The IP address is a 32-bit value when using IPv4 and is a 128-bit value when using IPv6. The IP address provides the information needed to get messages to the correct machine, but it does not provide any information about which process or application within the machine should receive the data. This information is provided by the network port number which is a 16-bit value used to identify the source or destination of network communication within a single host. Network ports allow processes to create software-definable communication points within a host. These points can then be used to send data over the network to another process with a corresponding IP address and port number.

When a process wants to send a message, it issues a call to the transport protocol. It is the responsibility of the transport protocol to handle the actual transmission of the data. This may involve breaking the message into smaller pieces, making calls to lower level protocols, and any number of other issues. All of these issues are handled transparently to the initial sending process. When the data arrives at the destination port, the receiver can issue a corresponding receive call to read the message from the network.

Transport protocols differ in the level of reliability that they guarantee to the user. Those that guarantee that all messages sent will arrive at their destination are called *reliable transport protocols*, while those that may only promise to make a best effort attempt to successfully deliver the message are called *unreliable transport protocols*. The TCP is a reliable transport protocol whereas the UDP is an unreliable transport protocol.

Message transmissions may fail for any of several reasons ranging from buffer overruns to faulty hardware. In the case a reliable transport protocol, the message will be resent until it is correctly transmitted. Unfortunately, guaranteeing reliability leads to extra overhead in the communication that can decrease performance. Because of this, many tools use unreliable communication to reduce the networking overhead.

2.1.4.2 Group addressing

The previous section discussed how to send a message to a single destination, a process called unicasting. But what happens when a process would like to send a message to many

destinations at once? It is possible to iteratively send the message to each individual destination one at a time, but this consumes more time and network resources than is strictly needed. Additionally, iteratively sending the data is potentially more error prone to implement. Fortunately, many transport protocols support group addressing methods that allow for sending messages to multiple destinations simultaneously.

Broadcasting One of the simplest and most basic group addressing modes to use is *broadcasting*. Messages sent using broadcasting go to all hosts on a specified subnet [Ste98]. This subnet is usually the local subnet of the sending host. The broadcasted message is received by all hosts on the subnet and can potentially be sent as a single message, thereby minimizing the traffic on the subnet. Broadcasting can be useful for situations where either every local machine needs to receive the same message or only the subnet of the destination node is known. The latter case is more generally called "resource discovery" and is one of the most widely used applications of broadcasting. Broadcast-based resource discovery is used in protocols such as ARP (Address Resolution Protocol), BOOTP (Bootstrap Protocol), DHCP (Dynamic Host Configuration Protocol), and NetBios.

Broadcasting can be applied successfully, but it has several problems that keep it from being widely used. The first problem is that it can only be used on a single subnet. Broadcasting is not designed for use with multiple subnets because it is not normally desirable to send messages to more then a single subnet. The second problem is that broadcasting dramatically increases network traffic and consumes more bandwidth than is needed for most group addressing situations. Broadcasting also suffers from the problem that it can not be used to address multiple hosts on a more fine-grained level. The message can only be addressed to everyone not to just a few members of a select group. To solve these problems we need a more feature rich group addressing mode.

Multicasting Another group addressing mode is *multicasting*. Multicast addressing allows messages to be sent to a set of destination nodes that can be located anywhere on the network. Nodes communicating using multicasting register their interest by joining a multi-

cast group. Each node that is a member of a given multicast group can then send a message to all the other nodes in that group. The message only transmits to the other members of the group. An advantage of using multicasting compared to unicasting is that multicasting can be more efficient in its use of bandwidth. Since the same message is being sent to each destination, the transport protocol can take advantage of routing trees to make sure the message is only transmitted once over a given communication link. This can greatly reduce the overall resources used to transmit the message and can allow all the messages to reach their destinations faster then sending them in multiple unicast messages.

2.2 Collaborative Virtual Environments

The term "collaborative virtual environment" can mean many different things. In the most general form it can be used to describe nearly any computer-controlled system that allows for collaboration. Unfortunately such a broad definition makes it very difficult to discuss CVE software in any concrete terms because it covers so many types of systems. This definition can apply to applications ranging from online web forums to MUD clients and to MMPOGs. Because this definition is so broad, will use a narrower definition. In our case we are going to examine the specific subset of CVEs that evolve from traditional virtual environment (VE) systems such as scientific visualizations, immersive VR, and gaming.

For the purposes of this research, we use Singhal and Zyda's definition of a *collaborative virtual environment* as a "software system in which multiple users interact with each other in real-time, even though those users may be located around the world ... These environments aim to provide users with a sense of realism by incorporating realistic 3D graphics and stereo sounds, to create an immersive experience" [SZ99].

The section sets out to describe further the specific aspects of CVEs and what issues are involved in their creation. We begin by describing the characteristics of a CVE. This is followed by a description of several perceptual issues and their relationship to CVEs. We then describe several technical issues and finish the section with a detailed description of one of the technical requirements of a CVE system.

2.2.1 Characteristics of a CVE

A CVE system differentiates itself from other collaborative systems by possessing the following characteristics:

- Shared sense of space

  A CVE gives all users the sense that they are not only in the space that makes up the virtual world, but that they are actually a part of the world. This space may be a room, a vehicle, or just an imaginary place. Whatever the space is, it should be presented to all users in a consistent way so that all users feel they are experiencing the same environment. Though the space does not have to be presented graphically, many of the most effective CVEs represent the shared space using an immersive three-dimensional graphical representation.[SZ99]

- Shared sense of presence

  Participants in a CVE should not feel alone in the environment. Instead they should perceive that they are sharing the space with other users. This feeling of being with other users in the shared space is referred to as having a shared sense of presence. This sense helps users interact with and relate to each other in the shared environment.

  In most current CVEs, users are represented using a virtual embodiment called an *avatar*. An avatar is a graphical representation of the state of a user in the environment. It is normally articulated with characteristics similar to a "real" human. These may include arms, legs, joints, and head. Although avatars usually take a humanoid form, this is not a requirement. Depending upon the specific environment, it can be better to represent the user's avatar using a more immaterial representation.

  An avatar gives the other users in the environment a way to know what other users are present, what they are doing, and how they are relating to each other. When a new user enters an environment, the other users can see the new user's avatar. When one participant is talking to another they can use the avatar to get an understanding of the current body language of the other user.

The avatar representation has routines running that control the current representation presented within the CVE. These routines create an avatar representation using all currently known information and sometimes use an avatar simulation to present a higher fidelity representation.

- Shared sense of time and interactivity

  A CVE needs to bring people together and support real-time interaction where all collaborating users see all interaction as it occurs. When one person makes a change everyone else immediately sees the interaction and the associated change. This also allows users to interact with each other directly. By allowing users to share these actions in real-time the system provides a shared sense of time for all users. It is this shared sense of time that provides a temporal framework for making decisions related to the flow of cause and effect within the system.

- Shared communication

  Communication is a cornerstone of collaboration. If users can not communicate with each other, it becomes difficult, if not impossible, to collaborate effectively. Although visualization allows for some shared communication, in most cases it does not provide enough on its own. Because of this, most CVEs have support for additional methods of direct communication. These methods may include communication channels such as speech, text, gestural, or video.

- Method of sharing

  As the previous characteristics illustrate, CVEs allow participants to come together in an environment to see each other, communicate with each other, and inhabit the same space at the same time. But for a CVE to allow useful collaboration, it must also allow the participants to share data within the environment and share control of this data.

  The data users share within the environment may range from engineering data to photographs. It is not the type of data that matters; it is the fact that collaborating users

want to share this information with each other. This shared information forms the basis of many of the discussions and collaborations between the users.

It is not enough to allow users to introduce new shared data; the users also need to share control of the information once it is part of the virtual world. Without shared control the CVE would be little more than a passive environment similar to a movie where all the users just observe the updates that other users are making. By sharing control, all users can manipulate and modify the information to actively participate in the collaborations. An example of shared control from the engineering domain is collaborative design reviews. When a new model is introduced into an environment, each user will want the change to manipulate the design is ways such as moving the model around of disassembling it. This type of shared control is key to many shared environment scenarios.

2.2.2 Subjectivity

Most people make the understandable assumption that everyone else experiences the world in the same way that they do. People also tend to assume that the world as we perceive it is how the world exists in reality. In both cases, these assumptions are wrong. In this section we will discuss these assumptions and how their existence can be used to our advantage when designing CVEs.

The perception of an individual person is based upon their personalized view of the world. This view is an internal representation of the world that is based upon the real world the individual is observing. Although based upon the real world, this internal representation may vary significantly from the real world being observed. Because of this disparity we recognize that there are actually two distinct world representations. There is the objective physical world of reality and the personalized internal world of the individual [**?**].

The objective world contains the true state of the physical world. The physical world is made up of objects that have a completely defined state. The state of an object may include things such as position, color, and any number of other attributes. This state is an absolute

definition of an object with no variability in the definition. An object attribute has a single quantifiable value and there is no room for interpretation. For example, the physical color of an object is defined by the frequency of the light waves reflected from the object. Given the same lighting conditions this value does not vary. It is constant and is presented to every observer in a consistent way.

The perceptual world, on the other hand, is very personalized because it is only represented internally within the minds of each individual. It is based off individual perceptions, judgments, and past observations. In most cases it reflects the objective physical world, but it is not an exact copy. This perceptual difference can be seen in the way each person sees color in a slightly different way. What looks like two vastly differing shades of orange to one person may look like a single shade of red to another person. As compared to the absolute object states in the physical world, all states in the perceptual world are qualitative and highly subjective. There is no single "correct" perceptual world model.

Because humans create their internal worlds based on individual perceptions, it is impossible to have exactly the same experience as another person. This can be seen by looking at the way two people in everyday life can both experience the same events but describe them differently afterward. They both saw the same things happen, but their perceptions varied based upon their personal perception of the situation. Each individual has a distinctly separate perceptual model of the world, and no two internal world models are exactly the same.

The variation in perception of the same situation from one person to the next is called *subjectivity*. Subjectivity can be defined as a differing in experience of an objective world due to factors internal to their perceiver such as personal preference, aesthetic sensitivity, cultural background, and so on [Pet99].

Subjectivity is also exhibited by users in virtual environments. When a user perceives a virtual environment, she does not necessarily perceive the current state of the environment. Instead, a user's perception of a virtual world is based upon a internal representation that is subjective. Their perceptions are based upon what they perceive as the current state of the environment and what they have experienced as the previous states of the environment. Even

if two different users are presented with exactly the same representations of the environment with the exact same changes, the user's perceptions are likely to be slightly different.

Another way to examine subjectivity as it relates to CVEs is as "equivalence of experience" [Fit99]. Each individual experiences the environment around them in slightly different ways. If two different people look at the same "red" object they may both perceive the color red differently. But the fact that there exists a set of common terms to describe these experiences allows them to communicate an equivalent perception. So although each individual may perceive a different color, they both describe it as red. As long as we can describe an experience in the same way to everyone else, then we will have all shared the same experience.

This equivalence of perception is the key to using subjectivity to relax the presentation constraints of a CVE. Because each individual user's perceptions of the environment is subjective and therefore not based exclusively upon the objective state of the system, the states do not have to match for each user. Our goal is simply to provide for equivalence of experience by allowing the perceptions of each user to be consistent.

Consider an example where there are three users A, B, and C in an environment. If user C hits a ball, both A and B should think that C hit the ball as well. If the presentation is slightly delayed for one user compared to the other, that is fine as long as they both have a consistent perception of the environment. As Fitch refers "both users end up with an equivalent interpretation of the virtual world and so can corroborate their perceptions without confusion" [Fit99].

We will discuss relaxation of presentation constraints in more detail later, but for now it is worth noting that without this ability it would be impossible to create true collaborative virtual environments.

2.2.3 Issues in Collaborative Virtual Environments

Crafting software for creating CVEs is much different from other software construction tasks. The task of creating CVEs involves taking into account many challenges and requirements that are unique to this type of software. By their very nature, CVEs must deal with

issues relating to human computer interaction, distributed systems, extreme scalability, flexible deployment, and heterogeneous execution. This section will go into the details of these issues. We will describe each issue, its impact upon the design and implementation of software for CVEs, and when possible we will describe common solutions.

2.2.3.1 Interactivity

The single largest challenge to developing CVEs is providing support for interactivity within the environment. If a virtual environment does not provide interactivity it will not be convincing. The users will lose the illusion of being a part of the virtual world and will instead see the environment as indirectly responding to their input.

To support distributed interaction effectively, a CVE must support two capabilities. First, users need to receive nearly instantaneous response to their interactions within the environment. When the user reaches out and grabs an object to move it, the user needs to think they immediately take possession of the object and are able to move it. If instead, the system pauses or blocks the action while processing some network requests, then the user loses the sense of direct interactivity. This first capability is needed to make the system look responsive to the local user. The second capability needed is that users quickly receive updates about interactions of remote users within the environment. This is important because it helps to allow multiple users to all interact with the same objects in the same environment. If remote updates are not reflected to the local user quickly enough, then the multi-user interactions taking place begin to break down due to perceived lag in the system.

As an example of the types of constraints that CVEs place on interactivity consider the situation shown in Figure 2.1 on page 32. The user would like to move a shared object represented by a ball. The ball's state is being managed by a remote machine, so the user's local machine can not directly update the ball's position. Instead a request is sent over the network to the remote machine. The remote machine processes the request and makes the necessary changes to the ball's state. Then this state change is sent to all machines in the CVE including the node that originated the change.

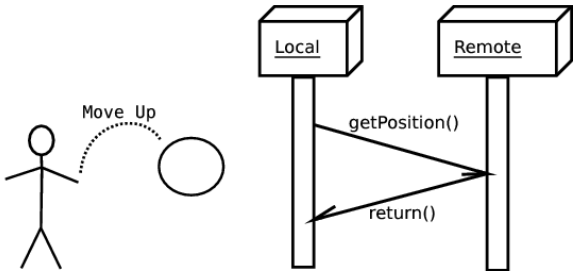

Figure 2.1 Grabbing a remote ball

Because the local machine needs to send a request to the remote machine and wait for the response, the results of the interaction are not immediately seen locally. If we assume that the network latency between the local and remote machine is L and that we initially attempt to move the ball at time $T_{initial}$ then the earliest that the user could see the response to the interaction is $T_{response} \geq T_{initial} + 2L$. So the amount of lag in response is equal to twice the latency of the network ($T_{lag} = 2L$). If this lag is too great, then the user will not find the environment sufficiently interactive.

In summary, the interaction lag in the system needs to be reduced as much as possible. It is not possible to remove all lag from the system, but the goal for any CVE system design is to reduce or mask the lag so the user feels that the environment is local. The system should make a best effort attempt to give the users the illusion of direct and immediate control over the environment. This will allow the user to see and react to the VE in real-time with no perceived delay between their actions and the effect of those actions on the shared environment.

2.2.3.2 Distributed systems and algorithms

Another challenge involved in implementing CVEs is dealing with the complexities involved in the distributed systems and algorithms needed by such a system. Implementing distributed algorithms is much more difficult than standard serial algorithms or multithreaded algorithms because developers have to account many extra design issues. These

issues include: network latency, limited network bandwidth, data loss, network failure, concurrency, and asynchronous communication.

We will now describe these issues in more detail referring to Figure 2.1 on page 32 as an example of a typical CVE remote operation.

Network Latency: Because distributed algorithms rely upon networking for exchanging data between the remote nodes, they must account for the communication lag introduced by network latency. Take as an example the case where the local node requests the current position of an object that is owned by a remote node. The request travels along the network until it reaches the remote node. The remote node then processes the request and sends a response back to the originating node. When this response is received it can then use the data in its local algorithm.

The difficulty in the example is that the local node must wait for the response message. This could mean that the local node is waiting for hundreds of milliseconds or more. During this time we do not want the local node to block waiting for the response. If it did this, then the local representation of the CVE would "freeze" and stop being responsive to the user. Instead we would like the local environment to remain interactive and responsive while the request is being processed.

To allow the local system to remain interactive, the request needs to be processed asynchronously. Instead of waiting for the response the local node should continue execution and at some later point in execution, be alerted that the response has been received.

Limited Bandwidth: The network links being used by the CVE have limited bandwidth. This is an issue in algorithm implementation because it limits the amount of data that can reasonably be passed between nodes as part of method calls or data updates. For example, if a CVE has a model of a car in the environment, it may be possible to send an update containing the position of the car many times per second, but it will not be possible to send the entire data for the car model as frequently.

Concurrency: The algorithms that control CVEs also have to deal with concurrent execution

of the code across multiple nodes. Every node in the CVE is simultaneously monitoring and updating the state of the shared environment. Additionally, some nodes may be executing simulation code for controlling the virtual world. This means that while the local node is requesting the position of the ball from the remote node, another node connected to the CVE may be doing the same or even updating the position of the ball. Keeping the state of the shared environment consistent requires that the implementation code function correctly in the presence of this degree of concurrency.

2.2.3.3 System heterogeneity

Another issue that CVEs must account for is client heterogeneity. The clients used to participate in CVEs can and do have vastly differing capabilities and resources. Each participant has different machines with varying local hardware, resources, and personalized configurations. A CVE must be able to run effectively on all of these different combinations of systems. Each client participating in the CVE session will have different computational, graphical, and audio capabilities. The CVE client software needs to take advantage of the local resources to provide the best presentation possible. This requires that the CVE can be presented in different ways at each client. For example, on one client the environment may be rendered using a very large number of graphics primitives, while on another client with less rendering ability, the world would be represented with less graphical complexity. To allow for this, the design of the CVE must allow for adaptability in local presentation and computation.

If the CVE client does not have the ability to adapt to local resource constraints, then all clients must present the world using the same representations. Because each client would use the same representation and perform the same computations, the experience of all users is degraded to the abilities of the lowest performance participant. This applies not only to graphical representation but also to network communication and interactivity. If each client is required to send out network change updates at the same rate, then the update rate of the system is limited to the network resources of the slowest system. So for example if there is a shared CVE with 10 participant where 9 users have high-speed network connections and

the remaining user has a much slower network connection, the experience of all users will be limited by the performance of the single low-speed connection.

Assuming that each participant has the same computational resources, each system may have different presentation and interaction methods available locally. For example one system may be completely desktop based with just a monitor and a mouse while another system may be an immersive environment such as a CAVE^{TM}. This presents a problem for CVE designers because the interaction and viewing capabilities of each system are widely varied. In the immersive system the user may have a positional interaction device that allows them to physically reach out and grab an object in the space. Using a desktop environment, the same user would have to use the mouse to move some virtual representation of their hand to grab the same object. This is potentially a much more difficult interaction and can lead to asymmetry in user interaction abilities. Additionally interaction differences can lead to the development of applications that either cannot be fully experienced or can only be partially experienced because of interaction limitations.

Interaction asymmetry can also lead to issues of fairness. When two users are both interacting with each other in the same environment but each of them has different interaction methods, one user may have an unfair advantage. This advantage could be caused by extra capabilities that they may posses or could even be caused by have less interaction abilities which could allow short cuts in interaction. For example if the shared environment is a game where a user needs to rapidly turn from side to side to see the environment (such as a first person shooter), then a user of a desktop system would have an advantage over a user in a fully immersive system. The reason for this is that the desktop based user is going to be using indirect methods to turn and walk. These methods are not restricted by physical constraints and will provide an advantage. The user in the immersive environment may not be able to turn entirely around in less then a second but the user of the desktop system may be able to turn around several times per second with no ill affects. Correspondingly, the user in the immersive environment will potentially find it easier to grab a virtual object then someone on a desktop environment would. This is because for the immersive user grabbing is a very

natural interaction; the user reaches out and physically grabs the object. The desktop user has to interact indirectly with a mouse to accomplish the same goal.

CVE developers must take system abilities into account to create worlds that can scale to any local system abilities.

2.2.3.4 Scalability

LSCVEs must deal with issues of scalability. Scalability is primarily measured in terms of the number of participating entities. These may be users, objects, or computer controlled agents. Theoretically there are $2^{num\ entities}$ possible interactions that could occur at any given moment. Managing this immense number of potential interactions is necessary to allow a large number of users to participate at any given time. If the clients are not able to interact with each other and immediately see the results of those interactions, the users' experience in the CVE will break down and effectively make the system unusable.

The interactions related to scalability are tied to the complexity of behavior within the environment. The degree of the inter-entity behavior and the complexity of that behavior actually determines the amount of interactions that must be supported in the environment. If all clients of the environment are simply viewing the current state and that state is determined solely by a central server, then the problem of scalability is greatly reduced. Conversely though, if every entity in the environment is simultaneously modifying the shared state and every client needs to see every update the problem of scalability become nearly intractable.

Fortunately the complexity of most useful CVEs falls somewhere in the middle of these two extremes. In most CVEs, the users don't all modify every piece of data and they are not all interested in the state of every piece of data every frame. Instead, collaborating entities form groups of collaboration. Within those small groups, each entity does operate upon the same shared data and each entity does need to see all the updates. To support a scalable system, the CVE software needs to recognize these groups of collaboration and use this knowledge to optimize the network communication within the system.

2.2.3.5 Deployment

Another issue that must be handled for CVEs is deployment. Deployment takes into account all the issues of how to get the client software and any associated resources to each of the client systems and configure the client to run correctly. Traditionally this had been handled using manual setup where it is the user's responsibility to make sure that they are running the correct version of the client software and have the current version of any other offline resources. This method can be rather tedious and error prone since users must be constantly vigilant to ensure they are running the correct software. It also causes problems for CVE developers because it is impossible to guarantee that the users always have the correct versions.

Recently CVE software has started to rely more on automated software updates. Using this method, each time the clients connect to the CVE, they automatically download any updated software and resources that are available. This removes the updating burden from the user and also allows the CVE to guarantee that the clients are all using correct versions of the resources. Configuring the client software still needs to be done manually, but even this can be eased by providing step-by-step tools and other methods to make it simpler to setup the systems.

It should be noted that automatic deployment suffers from the added complexity of heterogeneity and security. Heterogeneity introduces complexity because the automated update has to make sure to provide the correct version of the environment for the actual local hardware. If the local machine is using an Intel-based CPU running Linux, then providing it with an updated version of the software that is compiled for an Intel-based Windows systems will not work. The updating mechanism must be able to take any system specific issues into account when performing the update. Additionally, since it is downloading executable code, the updating mechanism must provide a method to ensure the update is valid and has not been corrupted or tampered with in any way.

The software architecture of the CVE system also influences deployment. If the software is packaged as a monolithic system, then each time the system is updated the entire distri-

bution needs to be updated and downloaded to the local host. If instead the CVE system is component-based, then it could be packaged as individual components. When a component is updated, then only that single component need be downloaded to the clients. Also, if the system is extended this extension could be captured in a single component for download. Because of their more modular design, component-based systems generally provide for more deployment options and are thus a good choice for LSCVE systems.

2.2.3.6 Failure management

In any software system, failures will occur. In a software system involving as many machines and complexity as a LSCVE, they not only occur, they occur very frequently. Failures can occur because network connection may become unstable either temporarily or permanently. Other failures occur because newly introduced user code may contain errors. Additionally, since each node may potentially have slightly different copies of the current code, there can be failures that occur because of differences in the underlying code itself. In a system as large and complex as a large scale CVE it is impossible to remove all sources of failures. Instead we have to do our best to design systems that can function correctly even in the presence of errors.

The types of CVE systems failures can be broken down into four major groups [SZ99]:

System stop: A system stop occurs when a failure in the system causes the entire CVE to stop functioning or crash. This type of failure may occur when centralized resource become unavailable such as a server in a client server based system. This failure can also occur if a corrupt resource is introduced into the CVE. This could cause all clients to crash and disconnect from the system. System stop failures are the most severe type of failure and as such are the least desirable type of failure. In general, a well-designed CVE should protect the system from system stops to prevent them from ever occurring.

System closure: A system closure occurs when the CVE system is still executing, but new clients cannot join the current CVE. There are several cases where this type of fail-

ure can occur, including failed centralized resources like an authentication server, scalability problems, and network failures. Active CVE users are not directly affected by this type of failure, but it can prevent the CVE from being useful since new users will not be able to join the session. This type of failure is serious, but if it is only a temporary failure the system can keep functioning correctly.

System hindrance: A system hindrance occurs when a failure causes the users' experience of the CVE to degrade. These types of failures can range from annoyances to severe problems. An example of a non-severe failure of this type is a single user becoming disconnected from the environment and thus disappearing from the CVE. The perceptions of the other users will be degraded because the other user disappeared, but the environment will still be running. There are more severe types of hindrances such as a system crashing that is controlling a simulation that many users are interacting within. In this case, the failure has a wide ranging effect because it keeps the CVE from functioning correctly for many users. In many cases, there is little that the CVE system can do to avoid system hindrances, it can only attempt to minimize their impact and prevent this type of failure from escalating into a more severe system stop failure.

System continuance: System continuance failures are failures that do not cause noticeable effects on the active CVE. This type of failure may include transient resource failures or more severe resource failures that can be automatically recovered from. An example of a low-impact failure would be a system monitor failing. Although monitoring may not be working, the users of the CVE will not notice the failure. A potential more severe error would be a server crash. If the system can automatically transition to a backup server, then this failure will be transient and user will not notice the change. But if the system does not support transparent transitioning to a new server, this failure could cause a system stop. The goal of any CVE system is to make it possible to transform as many failures as possible into system continuance failures.

2.2.3.7 Continuous operation

In the case of extremely large scale CVEs, there may be tens, thousands, or even millions of participants. Many recent massively multi-player games already maintain worlds with nearly one hundred thousand users participating simultaneously. In such large worlds, the CVE system must be in continuous operation. If the system has to be restarted each time a new software upgrade is needed to fix a bug or add a new feature this would be a great inconvenience to the users and would mostly likely lead the users to finding anther system that does not suffer from this issue. The system must instead support continuous execution where system components can be replaced, upgraded, and even added while the system is still executing.

There have been a few notable research efforts aimed at creating such a system. One of the leading efforts is the Bamboo project. Bamboo is based upon the following assumptions [Wat98, bam02]:

1. Eventually, there will exist a persistent virtual environment shared simultaneously by billions of participants.

2. There can never be a global reboot.

3. All modifications must happen on the fly.

The goal of Bamboo is to enable entirely dynamic scalable virtual environments to be always available. The design of Bamboo is based upon a plug-in component metaphor. Modules are loaded dynamically and can be retrieved from a local cache or remotely via protocols such as HTTP. Bamboo also supports the automated loading of module dependencies which are specified through an inter-module dependency specification method. Bamboo is designed to be cross-platform and to provide a common run-time for the modules to execute in. Bamboo aspires to solve a very complex and difficult problem. The project has done much to accomplish this, however at the time of this writing there is still much to complete. It is our hope that Bamboo or a similar project will be able to provide the underlying infrastructure for the next generation of persistent CVEs.

2.2.4 Shared state

As we previously stated, a key requirement of any CVE is to provide the users with a shared sense of space. A user feels a shared sense of space if they can meaningfully interact with remote users while being presented with a view of the virtual world that is consistent for all users. Because of subjectivity, world consistency does not have to be absolute. The worlds only have to be consistent enough to allow equivalent interpretations of the virtual world.

Providing sharing and consistency of the virtual world data is one of the core responsibilities of any software architecture for CVEs. CVE architectures normally provide sharing of the virtual world by implementing a data structure for sharing state. For the remainder of this section we will refer to this data structure as the shared state data structure or shared state for short. In most cases this data structure stores every element and attribute of the shared environment, including:

Object information: All objects in the virtual world store their full state information in the shared state data structure. Each object contains a mixture of common and custom object attributes. Examples of some common attributes include type, position, color, material properties, and geometric representations. Custom attributes can include anything a user defines for their application. In many cases, the sharing of objects with state is the primary method the system uses to track state and support user interaction.

User information: The shared state also includes information about who the current users in the environment are, where they are located, and the current state of their avatar. The user state includes information about the position of the user's limbs, the current direction they are facing, and nearly any characteristic that the environment may want to know about the user.

Algorithm information: The shared state may also include data that is specific to certain algorithms running within the system. This type of information is normally shared as object information, but differs in that there is not necessarily a visual representation of the information. The data is only shared to allow distributed algorithms to share state

> information. An example of this type of information would be a collision detection data structure for a set of objects in the environment.

Creating a method to share state information between nodes is one of the most difficult issues in CVE design. The method used can dramatically impact not only how well the system scales and how interactive the system feels, but also what types of applications can be run within the environment.

One of the reasons that this is such a difficult design problem is that there is a constant trade-off being made between realism and resource consumption. If the designers want all participating users to see every state change immediately after it is made, then the system must send out complete updates to every node each time a change is made. This can consume an enormous amount of networking resources and requires an ever increasing amount of bandwidth.

Additionally, if the designer wants to ensure that the state at each node is completely consistent at each moment in time, updates have to be synchronized. In a fully synchronized system, the node making the change waits to update its local state until every other node is ready to update. Then when each node is ready they all update their state at the same instant in time. This method of fully synchronized consistency dramatically increases system latency and correspondingly increases lag in distributed user interaction.

This trade off between realism and resource consumption is expressed in the *Consistency-Throughput Tradeoff* [SZ99, pg 103]:

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

As the tradeoff points out, a CVE designer either has to allow state to change frequently and deal with each node having slightly inconsistent state information, or they choose to keep a synchronized state that does not easily allow updates. This choice of how to deal with consistency versus resource consumption leads to many different ways to implement shared state data structures for CVEs.

There are three fundamental methods used for sharing state: central repository, frequent state regeneration, and state prediction [SZ99]. Most real world systems use one of these methods or a hybrid approach that combines features from each of them. The remainder of this section will describe each of these methods along with one example of a hybrid method.

2.2.4.1 Centralized repository

Centralized repositories maintain a consistent CVE state by keeping a single copy of the CVE at a centralized location. All users participating in the shared environment are presented with a view of this centralized copy of the data. Because all users are referencing the same copy, every user sees a consistent view of the environment. When a client needs to know the current state of an object in the virtual world, it must query the central repository for the current state. Clients may cache the state of the repository locally to increase the performance of reading the shared data. Updates to the repository are synchronized using locks at the central authority. The locking system used is responsible for guaranteeing that updates are performed in a valid order and that all clients receive the updates in the same order.

There are many widely used examples of centralized repositories outside the area of CVEs. Networked file systems are one of the most common examples. A networked file system provides each client with a consistent view of the current state of a shared file system. When changes are made to the files they are synchronized by the server using file locks. These locks ensure that all hosts sharing the file system see a consistent state for all the files. Other examples of centralized repositories include shared databases, web forums, and electronic scheduling systems.

There are many advantages to using centralized repositories for maintaining shared state. The primary advantages are that the programming model used is very simple and that data consistency is guaranteed. Since all object accesses are synchronized by a central server, the programmer has to handle very few synchronization issues. When they want to write code that updates the shared state, the code simply needs to acquire the object's lock, write the new data, and release the lock. This method is very similar to the standard multi-threaded

development model. The server handles the details of guaranteeing consistency of the data and the ordering of the updates.

Unfortunately there are some major disadvantages to using a centralized system. Since there is a single centralized server, the performance of the entire system is restricted to the performance of that single machine. The server machine presents a bottleneck in terms of processing power and network throughput. The server also becomes a single point of failure for the system. If the server machine crashes or becomes disconnected from the network then the entire system stops functioning.

Regardless of these major disadvantages, centralized servers are usually one of the first state sharing methods implemented by new projects. The reason for this is two fold. First they are relatively easy to develop because they involve no complex distributed algorithms. There is no need for a distributed consistency algorithm to synchronize data across machines because only one machine has a valid copy of the data. The second reason they are so popular is that they are very similar to many other client-server based systems in common use. Because of this, developers are already familiar with the issues involved in creating such a system. Thus the time to create and debug a working system in reduced and allows developers to get a system up and running quickly.

2.2.4.2 Virtual central repository

A very popular extension of the centralized repository approach is to use a virtual centralized repository. In this type of system, the shared state appears to be in a central repository, but there is no single central server. Instead, the repository is distributed across all the machines sharing the environment. The shared data is kept consistent using distributed consistency protocols. The various consistency protocols differ widely in their behavior based upon what the specific protocol is designed to optimize.

These systems inherit the same advantages as the centralized repository method, but remove some of its limitations. Because there is no single central server machine, there is no associated performance bottleneck accessing a single machine. Unlike centralized servers

where high load degrades the entire system, a virtual repository can scale to use all the computers in the network. Virtual repositories also help to alleviate bandwidth constraints since there is no central machine that all data must flow through. Instead the data distribution is shared among all the nodes in the system. Each node sends out updates for whichever data objects it maintains.

Virtual repositories also have much better support for fault tolerance. With a distributed repository, there is not central point of failure. If a machine fails the rest of the system will keep running. In many cases it is even possible for other machines to take over the responsibilities of the failed machine. This allows the system to work around many types of faults that would otherwise trigger a full system stop.

It should also be mentioned that nodes in a distributed repository setup do not have to maintain a copy of the entire shared state. If there are aspects of the shared environment that they are not interested in they can ignore those objects. Each node only needs to maintain the state of any objects that it is currently presenting to the user.

2.2.4.3 Frequent state regeneration

Centralized solutions provide a degree of *absolute consistency*. In its strictest sense, absolute consistency guarantees that if two viewers are remotely observing the same object at the same time it must have the same state. Because of network latency, centralized solutions do not normally provide for absolute consistency, but they are based on the idea that absolute consistency is the goal of the system.

Attempting to provide absolute consistency within a collaborative system is not always needed, warranted, or even possible. There are many objects in a CVE system for which clients do not require absolute state consistency. Consider an object that is moving through the sky such as a plane. If two separate clients draw this object at a slightly different location at the same time is there a problem? As long as the positions are relatively close to each other, the users observing the environment will perceive the behavior of the object similarly due to user subjectivity.

Systems and objects that do not require absolute consistency can use frequent state regeneration to share state. When using the frequent state regeneration method nodes periodically broadcast complete object updates. Each node has a set of objects within the environment that they control. Every node in the system periodically broadcasts the full state of all the objects under its control. Each client that receives these updates modifies their locally cached copy of the object's state to contain the new update. This allows the clients to always use the most recently received value to present the environment to the user

There are several different methods that can be used to decide how often to send these updates. Some systems send the updates whenever a change to the object occurs, others send updates at a predetermined rate. It is also possible to use varying rates for each individual object in the system. One commonality between all methods is that most require an object's state to be sent out at some minimum rate. This is needed so clients joining the environment can quickly observe the most recent version of all objects even if some of them are not changing.

This type of system provides several advantages. The first of these is that any node can join late and start observing the object updates. After a few update cycles they should have a complete view of the world. Frequent state regeneration also provide very low latency updates for the shared objects. Since the updates are normally broadcast immediately after a change occurs, there if very little that could be done to decrease the latency of the updates.

Another advantage of the system is that it can allow multi-user support to be added quickly to existing software. Existing software only needs to add support for broadcasting updates when changes occur and reading received updates into the local system state. Because of the simplicity of this approach, it has been applied to many single user applications as a first step toward basic collaborative support.

Unfortunately there are some severe limitations to frequent state regenerations. The most obvious is that it can consume an enormous amount of network resources. Since every update is sent to every node, the bandwidth used is very high. Another limitation is that latency in the updates can lead to user frustration and sickness. Since the data is always behind the current state, the users are forced to make updates to the system based on out-dated

information. When these changes turn out to be based on invalid data it can be difficult for users to interact in the environment. A more subtle disadvantage is that users are able to perceive differences in objects based upon their update rates and whether they are remote or local. Since each machine will be limited to sending updates based upon local hardware and network constraints, the objects will all be updated at differing rates. This difference can become perceptible and distracting to users. Even worse though is that since local objects do not suffer from the effects of latency, they will always be at a state in the future compared to other objects. This can lead to many artifacts in interaction and perception.

2.2.4.4 State prediction

The final method of state sharing that we will discuss is state prediction. This method is an extension of frequent state updates. State prediction is implemented exactly the same as the frequent state update method except that clients are allowed to approximate state between updates.

Clients approximate state between updates using interpolation. The client can use any of several algorithms to do this interpolation but the fundamental premise is the same; the client is using old data updates to approximate the current state of the object at the time that it is being rendered. One simple method that can be applied to physical objects is to use the laws of Newtonian physics. In this type of a system the velocity and acceleration of objects is either implicitly computed from the positional updates or is explicitly sent as part of the updates. Between updates the object's position is approximated using the basic laws of motion.

In addition to predicting object state, this method can also allow the system to be more restrictive about when to send updates. The goal of the update system is to only send updates when they are needed. If the number of updates sent can be reduced, this can reduce the amount of network traffic and thus increase the overall performance of the system.

One common way for prediction-based system to reduce the number of updates is to make use of the knowledge of the prediction algorithms being used. In this setup, the node controlling an object runs the same prediction algorithm on the object that the remote clients

are using. Based on this, the controlling node can track the difference between the predicted state and the actual state. If this difference exceeds some predetermined threshold then a change update is generated. For best performance, the threshold should also take into account how long it has been since the last update was sent and how long it will take for the new update to reach all the clients.

3 Related work

This section covers a variety of previous work related to this research. First we start with an overview of several of the most influential and widely used CVE research projects. Then we describe a few popular world model representations that have either been proposed or are currently in use.

This section does not describe non-VE collaboration tools such as chat rooms or instant messaging since these tools are outside the scope of this research. Where tools of this type are discussed in the following chapters we will directly reference and describe the particular software and features that we are discussing.

3.1 CVE architectures

Each of the following tools represents a development environment for creating CVEs. Each system has been used successfully to create CVEs and has contributed new methods to the field.

3.1.1 DIVE

Data sharing	Hierarchical entity tree
Programming model	All communication happens through the world database
Resource handling	Resources are requested from closest neighbor with the data
Networking	Peer-to-peer multi-cast. SRM for reliable multi-casting.
key techniques	peer-to-peer multicasting, sub-partitioning of world, loose consistency of data

3.1.1.1 Introduction

The Distributed Interactive Virtual Environment (DIVE)[FS98, VRM97b, Div03] is a distributed VR software toolkit developed at the Swedish Institute of Computer Science. The first version of DIVE was released in 1991 and at the time of this writing the most recent version is version 3.3. The DIVE software is a research prototype and is available in binary form for non-commercial use.

DIVE was also one of the first CVE research projects and has been used on a wide range of collaborative applications. DIVE is not just a communication library, but has been designed to be a complete application development framework. DIVE is of particular interest because it introduced several innovative CVE techniques that form the basis for many current systems.

3.1.1.2 World model

A DIVE virtual world is represented as a hierarchical database composed of entities. Entities in DIVE represent "objects" in the virtual world and define the object's graphically representation. There are many types of entities in the DIVE system. Each of these types is part of a entity type inheritance hierarchy that defines the relationship between the entity types. Figure 3.1 on page 51 shows an example of the entity types available in DIVE. In addition to contain-

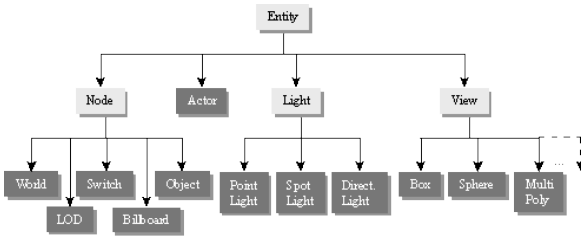

Figure 3.1 Entity types

ing the information and behavior specific to each type, entities can also contain user-defined data and even scripts for defining autonomous behavior. The hierarchy within the database is used to give an ordering and structural relationship to the entities as well as providing a structure for defining light-weight distribution groups which we will discuss later.

The software architecture of DIVE is based on a shared, distributed world database. The DIVE world database is a model of a virtual centralized repository as described in the previous chapter. Each node collaborating in a virtual world executes a separate application that uses the world database to communicate with applications running on the other nodes. In this way, client applications are not able to communicate directly with each other, instead all communication happens through the world database. The database is cached locally on each node to provide immediate access for the local application. Figure 3.2 on page 52 shows an example of five application processes communicating through two world databases.

Because applications communicate only through the world model, there is a complete decoupling between the application data processing and the distributed networking. This separation has several advantages. First it has allowed DIVE to change network back-ends without requiring application changes. A second advantage is that because developers write applications based only on the world model concept, applications do not require any modifications to work in a single-user environment.

When an entity is updated on the local system, the shared state is distributed to all other interested nodes on the network by sending an entity update message. The entity update message does not contain the entire state of the entity, but only contains the changes to the

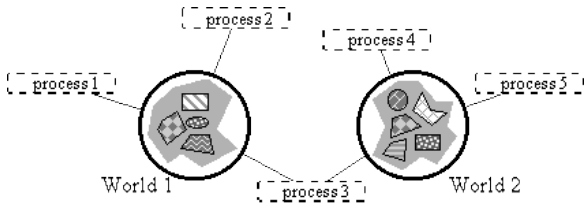

Figure 3.2 World and process model

object. Entity updates are immediately shown on the local system, but delayed on the remote systems until the update has arrived and been processed.

Any node in the system can introduce a new entities into a world database. Once the entity has been added, an associated update message will be sent to all other nodes in the system announcing the new entity's creation. If the new entity specifies a model file that a remote node does not have, then DIVE will automatically transmit the required data to the remote node.

All entities added to the world database are completely autonomous. This allows a node to join a world, add a new entity, and then exit the world while leaving the entity in the world. Since the entity has no owning node it can remain in the world database even after it's creator has exited.

DIVE supports world persistence, but it is not part of the core architecture. Instead, persistence relies upon either having a copy of the application always running, or having a client that saves snapshots periodically that are then used to start the system again from the saved state. It is difficult to implement reliable persistence in DIVE because there is no concept of a specific node being responsible for or owning an entity. This is an area that the DIVE developers have said needs improvement.

In addition to standard application data, DIVE also supports audio and video communication channels within the virtual worlds. These channels are not part of the world database. Instead, they are distributed using world-specific multicast channels.

3.1.1.3 Database replication

As we described above, the DIVE world database is cached locally at each node. These caches simultaneously support two forms of replication: active and partial

When a node makes a change to a local database entry, that update is sent to each other node. This form of replication is called *active replication*. It is called "active" because it is the responsibility of the node making the change to detect that a change has occurred and "actively" send out an object update.

Partial replication refers to two fundamental features of the Dive world model: loose consistency and light weight groups.

Loose consistency DIVE supports a form of loose consistency that allows the database models at each of the nodes to be slightly out of sync at any given moment. Because updates are processed by updating the local copy first and then distributing the update to all the peers, the update is applied at each node a slightly different times. As a consequence, the database is not always in the same state at each location. It also seems to be possible that two separate peers may update the same entity at the same time and send out conflicting entity updates.

DIVE tolerates these discrepancies in the world models and implements services that ensure their equality over time (using dead-reckoning and object update mechanisms). The DIVE developers reason that because most interactive updates are "bursty" there will only be short periods of time where many updates are performed. They have found that after such a burst of updates the state of the system will stabilize to some common state given enough time with no interactions. They describe this as stabilizing into a common "inertia" or lasting state. DIVE also implements algorithms to further assist the system in synchronizing by periodically synchronizing the data using sequence numbers to track entity version and possible update requests. The exact method used to implement this synchronization method is not described in the literature.

Light-weight groups DIVE provides the ability to divide worlds into sub-hierarchies that are only replicated between nodes that share interest. These sub-hierarchies are called

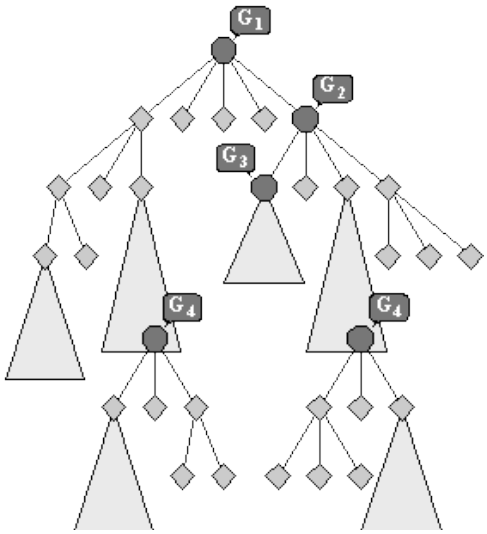

Figure 3.3 Light-weight groups

light-weight groups (LWGs). Applications that use LWGs can give DIVE extra information that keeps it from duplicating and monitoring sections of the database that are not of immediate interest to the application on the local node.

LWGs are defined by grouping nodes within the database hierarchy. Figure 3.3 on page 54 shows an example of a world database with LWGs identified by $G_{group\ number}$. So in this world database $G_1$is the base LWG for all communications in the system. The world database also defines three other LWGs that are used for this given virtual world. As the diagram shows, a LWG may be used in multiple locations within the world database. This is done where there are multiple parts of the world database that may be in the same interest group. When determining what LWG a node is in DIVE traverses up the world database tree until the first LWG node it encounters. This LWG is then the group for the given node. The topmost node in the database is the default LWG for all communication within that virtual world.

LWG are implemented by assigning each group a specific multi-cast channel. This multi-cast channel is then used to distribute all the data update messages for nodes in that group. Because light weight groups can be assigned to any multi-cast channel, it is possible for multiple light weight groups to be assigned to the same channel. This allows one interest groups to

receive updates for multiple LWGs and thus allows nodes to get updates for multiple LWGs very easily.

Because the sub-groups that define the LWGs are just part of the world database, they are not restricted to any specific spatial organization as in some other tools. Additionally since the assignment of LWGs can be dynamically changed at run time it is possible to dynamically change the interest management of the system while applications are executing. This combination of a very general specification and the ability to make changes dynamically allows developers to implement high-level interest management algorithms. This has allowed DIVE developers to experiment with many alternative area-of-interest management algorithms and allows application developers to tweak the interest management methods for the specific application that is executing[VRM97b].

The DIVE developers have found that the use of LWGs dramatically cuts down on network traffic since not all updates have to go to every node. Additionally it allows for increased scalability because applications only have to receive the updates that are of interest.

3.1.1.4 Network communication

DIVE uses peer-to-peer multicasting for network communication. Peers add themselves to a group by joining the associated multi-cast channel. Once a peer has been added to the multicast channel it can send message to all other connected peers.

DIVE supports both reliable and unreliable multicast communication. Reliable communication is implemented using the Scalable Reliable Multicast (SRM) protocol[FJL+97]. This protocol uses negative acknowledgments and a request/response scheme to allow reliable multicasting of data. Unreliable communication is implemented using standard multicasting.

DIVE only uses two basic message types. The message types differ in the type of data being sent and the reliability needed for the data transfer.

Message types:

- Entity updates and database modifications

Communication method: Reliable multicast.

All entity updates and database modifications are sent on multicast channels as defined by the LWG containing the given entity. Although the transmission is reliable there is no guarantee of message ordering.

- Continuous data streams (audio, video, etc)

  Communication method: Unreliable multicast

  DIVE maintains a separate set of multicast channel per world for real-time data streams such as audio and video. All streaming data associated with the virtual world is transmitted on this channel. The data transfer is unreliable to minimize latency and ensure continuity of data (not consistency). The fact the entire virtual world shares these channels causes scalability issues with the current implementation.

When attempting to find an entity, a query is sent out on the relevant multicast channel. This query uses an algorithm supported by SRM based on round-trip-time estimation and a timeout to find the closest peer with the latest version of the object responds. Additionally there is multicasting to prevent the query from receiving similar response from other peers.

3.1.1.5 DIVE name server

The DIVE name server is an application that helps clients find and enter available virtual worlds. A client connects to the DIVE name server with the name of a virtual world that they would like to enter. The server responds with the current multicast channel for that world. The name server is only used in this initial communication and is not involved with any further communication in the system.

If the connecting client is the first to connect to the virtual world, then it is responsible for loading the initial state of the world. If it is not the first, then it uses a round-trip time estimation algorithm to find the nearest node likely to have a copy of the world. It then requests the initial state of the world from this node and begins loading the initial state before executing the application.

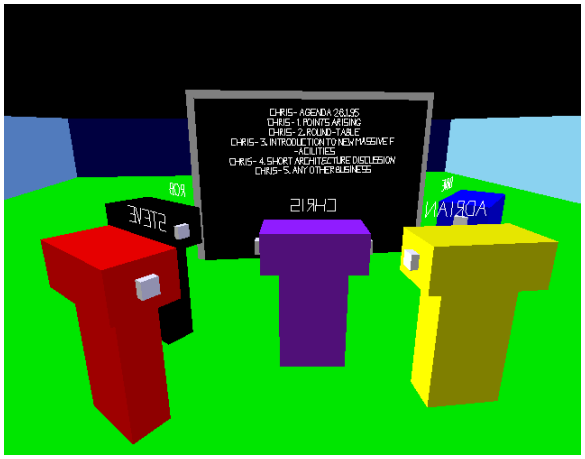

Figure 3.4 Example of a MASSIVE application

3.1.2 MASSIVE

Data sharing	Direct communication
Programming model	Direct communication
Resource handling	Application dependent
Networking	RPC and streams
key techniques	spatial-trading, direct communication

3.1.2.1 Introduction

MASSIVE is a research project from the University of Nottingham[GB95b, GB95a, BBFG94, BF93, BG97]. MASSIVE was developed as a testbed for several innovative interaction models and was developed to support up to 10 users in a shared environment. It as not designed to be a general-purpose VR application development system but is only designed as a teleconferencing system. We still include a description of it here though because it has influenced the design of many current CVE solutions and it forms the basis of the later MASSIVE3 work which we will cover in the next section.

3.1.2.2 System architecture

Massive is based upon the concept of a set of processes that communicate via typed connections. These connections combine RPCs, attributes, and streams. Massive does not use any shared data structures or databases but instead relies upon all processes to communicate directly with each other using peer-to-peer interfaces.

This communication method was chosen because it support communication contexts between objects. When two objects within MASSIVE need to communicate, they form a direct connection from one to the other. The communication connection provides a context for the communication. This context simplifies collaborative communication because both ends of the communication have an unambiguous view of the current state and purpose of the communication. This is an advantage over the distributed systems based on shared databases because it allows for direct associations between objects and nodes.

3.1.2.3 Spatial trading

The most unique capability of MASSIVE is its spatial trading model of interaction. This model associates an aura with each object in the world. An object's aura describes the spatial extents around an object where interactions are allowed. The system has an aura manager that tracks the auras of all the objects in the system at any given instant. If the auras of two objects collide (overlap in space) then the objects are said to have an aura collision. The aura manager is responsible for detecting these collisions and notifying the objects involved that a collision has occurred. These colliding objects are then able to communicate and interact using the objects' interface(s).

Objects have one or multiple communication interfaces. These interfaces are what objects used to communicate with each other. For one object to communicate with another, it must first acquire a compatible interface to the other object. Objects request these interfaces from an interface trader. An interface trader is a central location that maintains a list of all the available interfaces for all the objects in the system. When two objects have an aura collision and would like to communicate, they query the trader for compatible interfaces. If there is

a compatible interface available, then the trader will return it along with a new peer-to-peer connection and associated communication context. This connection in the typed connection that allows objects to communicate and interact

Objects do not initially know about any other objects in the system. They rely upon spatial trading to discover the other objects in the environment. When an object enters the world, it contacts the aura manager to register its interfaces and the aura's of each interface. Only when the aura manager notifies the new object of an aura collision does the object learn of the other objects within the system.

Objects can communicate with objects of other types and objects of types that were previously unknown. This is allowed because all objects dynamically register all their interfaces with the aura manager. Because the aura manager takes care of hooking up compatible communication channels during aura collisions, the objects do not ever have to communicate outside of the pre-specified interfaces.

The spatial trading model is used to realize two main objectives: scalability and controlling spatial interaction

Scalability is facilitated by limiting the number of object interactions that need be handled. In the case of Massive, this is limited by the object auras and the aura manager. Since an object only connects to objects that have aura collisions, the number of interacting objects is greatly limited. This correspondingly limits the number of network connections needed and the amount of network resources consumed.

Spatial interaction is controlled using the concept of awareness. An object's awareness of another object in the system specifies the relative importances of that object. The more attention given, the more resources will be devoted to that object's representation in the virtual environment.

Awareness (and auras) are influenced through the concepts of focus and nimbus. Focus quantifies the observer's allocation of attention. Nimbus quantifies the observed object's observability or the degree to which it is interesting. By combining these two aspects of the interacting objects, the spatial trading mechanism arrives at a decision about the relative im-

portance of interacting objects. If the relative importance is high then more resources are allocated to those interactions. More details about this innovative interaction model can be found in the published papers[BBFG94, BG97, BF93].

3.1.3 MASSIVE3

Data sharing	Object-based shared data service
Programming model	Agent processes communicating through shared environment data structure
Resource handling	Resources are part of application
Networking	Reliable and unreliable multicasting
key techniques	HIVEK consistency model, Agent processes, event queues and filters

3.1.3.1 Introduction

MASSIVE3 is the 3rd generation of the MASSIVE Project. It builds upon the experiences of the team developing MASSIVE1 and MASSIVE2 as well as shared data consistency work from Reading University[RS97].

MASSIVE3 is based upon the HIVEK project. HIVEK provides a shared data system that was created specifically for the needs of CVEs. In addition to the shared data service, there are other related services for supporting CVEs. This includes a naming service, simple rendering, and interaction toolkits, basic networking, object serialization, and distributed audio capabilities [Mas, Gre99a, Gee00, Gre99b].

3.1.3.2 Agents and Environments

The HIVEK shared data system is based upon two fundamental elements: *Agents* and *Environments*. Agents represent one main thread or process in the system and environments encapsulates the concept of shared data. When the system is running it consists of many agents communicating using one or more Environments. The Agents can be running on any

number of different machines that are all sharing the distributed environment(s). All inter-agent communication is done through the shared environment data structure.

HIVEK environments can be thought of as a shared database of the "environment" of a virtual world. Environments are structured as a tree (potentially only a partial tree) of typed data. The tree structure is similar to a scene graph without the geometric detail. The basic data types that are available include transformations (entities), geometry files, switch nodes, and text attributes. There are also more advanced data types such as object trajectories, event filters, boundaries, links, and update requests.

Agents access and modify the Environment using an Environment API. Agents use the Environment API to read items from the environment and iterate over the entire contents of the environment. To modify the environment an Agent uses the Environment API to create an update event object. Event objects are used to represent all database changes. The event objects supported by the system include: add, update, and delete.

When an Event object is created it is immediately placed into two queues, the sending and the pending queues. The sending queue contains change events that need to be sent to other nodes in the system. The pending queue contains change events that have yet to be applied to the local Environment. The queues are both flushed periodically when the system is ready to process the events.

One ability the event queues provide is that they can be modified to customize the behavior of the system. Custom event processing models can be implemented by Agents that directly access and update the event queues. Additionally developers can add filters the event queues to modify the way events are processed in the queue and passed on to the rest of the system.

In addition to the event queues, HIVEK also maintains a system clock for each environment. This clock is synchronized to a central clock and allows event objects to specify time constraints. The clocks are synchronization using a periodic time-stamp exchange method.

3.1.3.3 Sequencing

Total ordering of operations on data items is implemented using sequencers composed of a two-part numeric sequencer. The first value is incremented when a reliable update is made (this includes ownership transfers and deletions). The second value is incremented when an unreliable update is made and is also reset to 0 when the first value is incremented.

The sequencer is also used to enforce event sequencing. Event sequencing enforces that all mandatory updates are applied in order in every environment which handles them. Alternatively, optional updates are ignored if they are older then the current sequencer but they are still never applied out of order. The sequencer is part of any ownership transfer to guarantee that all pending updates are applied correctly at the new controlling node.

The sequencing system also allows for a very flexible method to specify addition operation ordering. Each update event may specify a set of events which must precede it (using item sequencers). Normally this only includes the standard sequencers. But by specifying dependencies upon other items sequencers, the sequencing of changes can be made explicit within the shared data structure. These explicit dependencies can be used to implement a variety of constraints including causality constraints.

3.1.3.4 Data consistency

HIVEK has very flexible support for distributed data consistency. The method used is a conservative method based upon a single transferable ownership per data item. All data items have an Agent that owns them and a data item can have only one owner at a time. Requests for ownership transfers are explicit events within the system and can occur at any time. The processing of ownership requests is based upon the current locking state.

HIVEK supports three different data locking methods – soft, hard, and control. The naming of these lock types is based upon how they respond to and process ownership requests. Soft-locks give away ownership upon receiving any ownership request. Hard-locks and control-locks never give away ownership upon receiving a request but may perform other processing on behave the requester.

Control-locks differ from hard-locks in that they signify to other agents that the owning agent may respond to remote update requests on the object. An UPDATEREQUEST is an EventObject that requests the owning agent changes the value of an object to a requested value. Upon receiving an UPDATEREQUEST the controlling Agent can choose to honor the request, ignore it, or modify it depending upon any local constraints it wishes to impose on the data item.

Centralized updating By making use of update requests, HIVEK allows a form of centralized updating. There are two supported forms of centralized updates (one for unlocked items and another for locked). In both cases instead of updating the object directly, an Agent requests an update by creating an UPDATEREQUEST event which includes the new desired state of the item. This is considered a centralized approach because the requesting Agent never acquires ownership of the object. Instead they treat the current owner as a centralized "server".

When an UPDATEREQUEST is received by the owner of an unlocked item, the current owning agent will apply the change and generate a new update event. If the owner receives multiple UPDATEREQUESTS, the updates occur in the order that they were received.

When an UPDATEREQUEST is received by the owner of a control locked item the request will not be automatically applied. Instead the owner may apply additional constraints to the request and then apply it or ignore the request entirely. For example the owner may restrict the position of an object based on some physical constraints in the environment. In this case, the UPDATEREQUEST is ignored and the owning Agent creates a new event that partially implements the requested change.

Ownership transfer methods When updating items using the non-centralized approach, the system must transfer ownership between the Agents in the system. The normal method of ownership transfer is a need-based transfer. When an Agent needs to update an object but does not currently have ownership, it requests the ownership from the current owner in the system. If the ownership request is satisfied, then the Agent can update the object imme-

diately otherwise the change request fails. Even if the Agent requesting ownership has the request granted, it must wait for a minimum of the round-trip delay time in communicating the request with the current owner. This time directly impacts the perceived interaction latency in the system and should be decreased if possible.

An alternative designed to decrease the latency is to combine the ownership request with the first data update. Upon receiving this type of ownership request, the current owner can choose to send out the first data update immediately and then transfer ownership to the requestor. This allows other Agents in the system to see the pending update faster then in the normal on-demand ownership request method.

Another alternative for reducing latency involves predicting ownership requests. If an application can accurate predict the need for future ownership, then the it can request ownership before actually needing it. This method relies upon application specific prediction methods, but when applicable it can greatly reduce the perceived latency in the system.

The primary disadvantage of using a predictive approach is that as mis-prediction can be expensive. An unneeded ownership transfer can waste system bandwidth and resources, but more importantly it can actually increase latency in the case where another Agent needs ownership of the object.

Advance communication When future changes in the system are known ahead of time (ex: a deterministic change), it is possible to make use of advance communication. Advance communication sends the change event with an associated minimum delivery time. The change of the item is delayed until that delivery time is reached then the change is enacted in all Environments in the system. Advance communication can be implemented as an additional causal constraint on the event object.

Wait for ownership mode The Environment API also supports a "wait for ownership" mode that accumulates change requests until an ownership change takes place. This allows the system to proceed without waiting for an ownership response to be processed.

3.1.3.5 World structure

Environments are represented using locales. A locale specifies a local coordinate system and may represent a region in the virtual world (ex: room, building, world). Locales can have boundaries which are methods of linking to other locales in the system. These boundaries allow locales to come together to form larger virtual worlds.

In addition to having a main environment for a locale, there may be other environments that represent a partial or simplified variant of the same locale. These variants may be automatically or dynamically updated. None of the associated documentation we have found describes exactly how these additional environments could be used but it is an interesting idea none the less.

3.1.3.6 Application development

Applications are written using the Environment API to create, manipulate, and monitor items within the environment. The development model is based on responding to events within the environment. For example responding to the presence of a user by updating the state of an object in the environment.

User interaction is based primarily upon mouse interaction. The viewer uses a mouse to support basic navigation and gives feedback on current state (i.e. direction, movement constraints, etc). The user is expected to interact with the application using direct manipulation of virtual objects. For example, moving the mouse over an object, clicking on it, and then dragging it to a new location.

3.1.3.7 Persistence and logging capabilities

Persistence is supported by application convention. "Well-behaved" applications start by loading their initial state from a file on disk. During execution, this file is periodically updated with the current state of the application. When the application has been unoccupied for a predetermined amount of time, the application should save the current state to the file and exit. The application then has to be restarted to reload the world. To support easier

startup, MASSIVE3 has support for automatically doing this in response to a name server request from a new client.

The system also supports event logging. All events for a locale can be recorded to a disk log so they can be played back later to recreate the environment edits. The system supports replaying these events so they can be paused, time varied, reversed, and restarted at any point during execution.

3.1.4 Deva

Data sharing	Object-to-object communication
Programming model	Communicating entities using message passing
Resource handling	Resources are located on the server
Networking	Reliable networking
key techniques	Subjectivity and environmental entity properties

3.1.4.1 Introduction

The Deva project is a framework for distributed VR applications[PCMW00, Pet99, CP01]. It was created at the Advanced Interface Group and the University of Manchester by Steve Pettifer. Although it can apply to VR in general, its main uses have been for desktop-base collaborative VR systems.

3.1.4.2 Metaphysical

Deva focuses on the metaphysical representation of the virtual world. Deva proposes that the world should be represented using a "metaphysical framework". This framework is responsible for providing a mechanism for extending, altering, and managing the properties of objects and the world around them. Additionally the framework should allow the modeling of the effects that the objects and the world have on other objects within the world.

Therefore its main focus is on how to represent the fundamental compositional aspects of the virtual world. This includes how to represent entities in the virtual world and how to represent both the common and unique properties of those entities. Deva also includes methods

for representing universal laws or constraints on the entities in the world. For example how to model a rock and the fact that a rock has mass that the law of gravity governs to make the rock fall.

3.1.4.3 Structure

- Presentation
  - Client view of the environment
  - Subjective presentation (perception)
- World model
  - Underlying world simulation
  - What is going on "within it"

3.1.4.4 World model

The Deva world model is "based on a particular interpretation of our experiences of the everyday world". The world model is based on breaking the world into *environments* which contain *entities*. This corresponds to standard terms of "places" that contain "things".

The language of Deva "embodies the idea that many of the properties of an object in our real world can be seen as being the result of an interaction between the object and its surroundings".

The Deva system focuses on the concept of a world model as opposed to a programming model. In the Deva world model, the world is represented as a pool of inter-related entities. The combination of all the entities in the world model form the actual environment that the user perceives. In this model, applications are simply represented as sub-sets of the world model made up of application specific entities.

Entities can represent objects in the virtual world, properties of an environment, or abstract programming concepts. The entities are objects made up of attributes and a callable

interface. The attributes hold state information about the entity. The interface defines a set of methods that other entities can call to communicate with the entity. These methods are primarily used to manipulate the state of the entity held in it's attributes.

Deva allows entity attributes to come from multiple sources (for example from the object and from the room the object is in). Deva implements this using entity property chaining. This is a technique where by entity properties determine authority and constraints on the entities.

A unique capability of Deva is that it allows entities to be extended dynamically with new attributes and behaviors. Deva encapsulates a collection of methods and attributes related to a single concept into a *component*. Deva can graft these components onto an existing entity to add new capabilities to already existing objects.

Deva defines several sources where entity definitions and extensions can come from.

Innate attributes and behaviors are entity aspects that are always tied to the particular entity type. These are the characteristics of an object that are normally thought of when defining a new object. For example a ball always has a radius and a color.

Inherited Entities can also inherit behavior and properties that are common to a group of entities. These are similar to innate aspects except that they are shared by a common group of entity types. An example of this may be that all physical objects have an attribute that defines their current position in the world. This attribute could be defined in a common entity type that all other entities inherit from to get this attribute and associated interface methods.

Imbued Entities can have behavior grafted onto them by the environment they are within. This allows the environment that an entity is within to influence the characteristics of an entity. For example an object that enters a Newtonian world environment may have a component grafted onto them that defines the attributes of mass, velocity, and acceleration as well as their associated interface methods. This component then forces the entity to "behave" correctly based upon the laws of Newtonian physics.

The ability of the environment to influence the behavior and definition of an entity is one of the most innovative aspects of the Deva system. It provides a potential solution to the very difficult problem of how to effectively define rules and constraints that govern specific areas of a virtual world.

3.1.4.5 Execution environment

The execution environment of Deva integrates entity behavior and interaction code within a single environment. Within this environment, the programming model is based upon communicating entities. All entities can retrieve and exchange references to other entities within the system. Using these references, entities communicate remotely by calling the methods in the entity interface. This communication system is based on an internal message passing system that makes the entity communication location-transparent.

The physical distribution of the code in Deva uses a client/server model where the application logic code executes on the server cluster. This server cluster is responsible for executing and maintaining the state of all entities that make up the virtual world. Because of this, the entire simulation for the virtual world is executing only on the server cluster.

Clients applications only have a subjective view of the current environment. They do not directly execute the code of the virtual world, instead, the clients have a "subject" object that mirrors each entity. This subject represents what the entity looks like and provides a placeholder that the client can use for interaction. All interaction with the subject view of the entity is reflected back the the server for processing. This indirect method of entity modification and interaction can add latency to the system, but simplifies the programming model used for entity-to-entity communication and processing.

Users are “simply considered to be an entity whose behavior is controlled by input devices rather than algorithms”. They are represented by standard entities with one entity per user in the system. The user entities have actions and methods designed to be controlled by input devices, but otherwise it is just a normal entity in the system. To interact with the system, a user updates the state of their associated user entity subject on their local machine. These

changes are then passed on to the server cluster which adds the changes to the currently executing simulation.

Applications as such are not directly represented in Deva. Instead, applications are "represented by an entity, environment or collection thereof that affects the behavior of other entities or environments". This leads to a rather unorthodox programming model that can be difficult to use at first.

3.1.4.6 Access Model

Deva has a prototype of an object access model. An access model is a "set of mechanisms to determine the operations that may be carried out on an object by a given user". Please note that these are not security models (this would enforce an access model).

Access models are needed in CVEs to support object ownership and authority. Access models allow objects to be intelligently managed to make sure that only those users that are authorized to make use of an object in the environment do so. The access model also allows different applications to exchange objects in intelligent ways.

The Deva model allows the access control to be constructed by the programmer and attempts to provide an interface that allows users to interact with and understand the access control system.

There are several components of the Deva access Model.

- Entity: A general resource (standard Deva resource)
- User: An entity that is related to a person
- Key: Entity describing dynamic relationship between user and entity
- Key manager: List of pending requests for validation. Returning new payload for compound rules.

The mechanism used is based upon validating entity method calls. All system users have to be validated to use the system. Once they are validated, the system key manager can provide method keys to authorized method calls.

This leads to the following sequence of events when a method call is made. The first step of any method call is to query the key manager for the current keys associated with the given user and the entity they are trying to call. This key is then passed as part of the method call mechanism. Before processing the method call, the system validates that the key provides the correct level of access to the entity. If it does not provide access, then the method call is ignored.

3.2 World representation

In addition to directly programming a CVE, there are other methods for creating CVEs based on specifying how the world is represented. These specifications do not specify how the underlying code should implement the sharing and viewing of the world. Instead they simply describe the virtual world in a way that should allow a variety of possible viewers to be development.

3.2.1 VRML

The VRML specification defines a method for describing virtual worlds. This section discusses version 2.0 of the VRML specification commonly called VRML 2.0 [VRM97a, HW96].

VRML was designed for the creation and distribution of web-based virtual environments. VRML was one of the first efforts to create a standard method of representing virtual worlds in a way that they could be viewed and used on the Internet as an extension of the world wide web.

VRML files are viewed using a VRML browser. VRML browsers are customized for viewing and interacting with VRML worlds. They are able to load VRML files, parse the contents into an internal representation, and then present the world to the user. The user can use the browser to move through the virtual world and to interact with the environment using normal desktop input devices such as a mouse and keyboard.

The core of the VRML specification a declarative file format for virtual world representations. This format is derived from the format used by Open Inventor [SC92] to store scene

graphs. This format is especially convenient because VRML worlds are represented as scene graphs. The scene graph for the world is composed of a hierarchical set of nodes. There are nodes for representing physical objects such as spheres, cones, and lights. There are also other nodes used as containers and structural nodes. There are also some special nodes defined for advanced features such as hyperlinking and prototyping.

VRML nodes contain fields which hold the data of the node. Each field has an associated type that defines not only the type of the data, but also the cardinality. For example a sphere node would have a field of type SFFloat (a single float value) for radius and a field of type MFFloat (a multi-value float) for center.

3.2.1.1 Programming Model

VRML2 programming is based upon events and routes.

Most nodes contain events. These events are a indication that something has happened related to the object – for example a field value has changed, the user has selected the object, or a timer object has expired. There are two kinds of events in the system: incoming events (inEvents) and outgoing events (outEvents). To enable event processing, the outEvent of one node is wired[1] to the inEvent of another node. When a field sends an outEvent the value of the event is sent to any inEvents that are connected to it. Upon receiving an event, the inEvent is set to the value of the sending outEvent that triggered it.

This simple programming model allows for a large degree of flexibility in development. Most of the programming within a VRML world is based upon animation and response, but there are facilities for more general purpose types of programming. This more general programming makes use of special scripting and sensor nodes that are specifically designed for creating interactive worlds.

Scripting is supported through script nodes. These nodes are just like other nodes in that they receive incoming events and generate out going events. They are unique though in that the node contains code for processing the incoming events before deciding what out going

[1] VRML uses the term wired to refer to connecting an outEvent to an inEvent.

events to generate. Developers can specify as many incoming and outgoing events as they like.

There are several stages in a normal VRML application interaction. First there is a trigger. A trigger is an item in the world that generate the event that starts the interaction. In most cases, the trigger is a sensor that sends an event based upon user interaction. The next stage is logic processing. This stage involves running script code to process the trigger event and potentially perform extra calculations before deciding upon a final event to output or action to take within the system. In most simple interactions this stage is not needed.

The next stage is a processing stage. In VRML documentation this is commonly broken up into two separate stages based upon timer nodes and engine nodes. The processing stage is responsible for taking the output from the trigger or logic and then controlling some final output that is used to update the scene. The processing stage allows the system to have a form of continuous processing that is triggered from the originating event. The timer node combined with other processing nodes can be triggered to keep updating the environment for some set period of time.

The final stage is the target. The target is the final node whose fields are being changed by the given interaction. Once the field is updated, the node will reflect the new information in the next rendering.

It is important to note that the communication between each stage of the interaction programming is still just events and fields. This is partially limiting in the way programming can be done, but it provides for a uniform and flexible interface between the components in the system.

3.2.2 Living Worlds

Living Worlds (LW) is a specification of a proposed standard for distributed object interaction in VRML 2.0[Liv03]. The goal of LW is to define a set of VRML 2.0 conventions for supporting environments that are *interpersonal* and *interoperable*. The LW group defines interpersonal as meaning "applications which support the virtual presence of many people in a

single scene at the same time: people who can interact both with objects in the scene and with each other". They define interoperable as meaning "that such applications can be assembled from libraries of components developed independently by multiple suppliers and visited by client systems which have nothing more in common then their adherence to the VRML 2.0 standard."

The LW specification address the following issues[Liv03]:

- coordinating the position and state of shared objects
- information exchange between objects in a scene
- personal and system security in VRML applications
- a library of utilities and some workarounds for VRML2.0 limitations
- identifying and integrating at run-time interaction capabilities implemented outside of VRML and its scripts

It is important to note that the LW group is not trying to create a new standard but is trying to evolve the current ones to add support for CVEs. Because of this, the LW specification builds upon VRML 2.0 instead of trying to create new mechanisms. In a similar fashion to VRML, the LW specification does not specify designs. LW only proposes a standard for how to describe collaborative spaces. Application developers are free to use any design to write applications that can view worlds specified using the LW specification.

The LW group believes that shared CVEs will not be designed as single coherent environments, but will be made up of components that are combined ad hoc as users extend and move around in the system. These components may not have been designed together or even tested together before they must interact with each other at run-time in the system. Because of this the components will need to have standard interfaces that allow them to work together.

The LW specification is proposed to be this standard interface between the inter-operating components. The specification will allow shared environments to be constructed by specifying them using VRML and associated extensions. This specification allows all the components

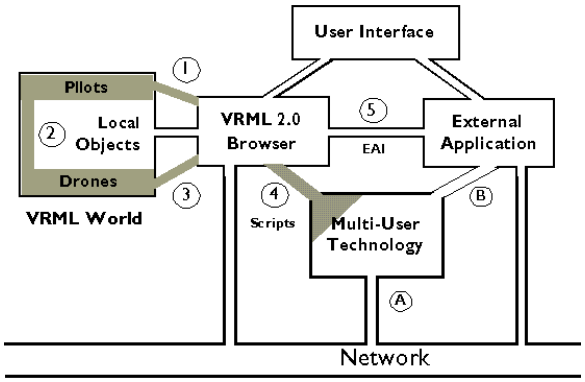

Figure 3.5 Living Worlds model

in the system to interface with each other to form a coherent shared world. The LW group believes that the evolution and creation of a shared cyberspace relies on the creation of reliable infrastructure.

scene sharing: Scene sharing captures the infrastructure for coordinating events and actions within the shared environment. Scene sharing allows multiple clients to interact with each other and make modifications to the current scene. It also provides the support for allowing objects to communicate with each other and exchange data.

security: There are many potential security problems introduced in large distributed systems. The security requirements for LW include concurrency control, conditional access, and user authentication.

3.2.2.1 Conceptual framework

LW specifies a general conceptual framework for implementations of the standard. The framework does not specify a design, it is only meant to be a guide for how to logically partition the system functionality into conceptual components. The standard can then define the interfaces between these components to allow the actual implementation components to come from independent suppliers.

The LW conceptual framework is show in Figure 3.5 on page 75. The components of the framework are:

User Interface This component accepts all user input and renders the virtual world. This component handles mouse, video, and sound in addition to the standard windowing system details such as widgets, menus, and dialogs. It provides all methods provided by the system to allow it's user to interact with the system.

VRML browser This component is responsible for loading the VRML world and providing an interface that other components use for all changes and updates to the run-time world.

World VRML virtual world which defines the scene and any associated script nodes.

External Application any external code or application that interacts with the VRML browser to provide some external functionality.

Multi-User Technology (MUTech) a component proposed by LW to implement sharing across the network. This component provides all the networking facilities needed for multi-user interaction beyond that provided by the browser.

3.2.2.2 System architecture

LW builds on top of the structural relationship mechanisms already present in VRML. These mechanisms consist of Nodes, Routes, Events, and Scripts. Within these mechanisms, the LW specifications sets out to define methods for declaring the relationships of a shared virtual world.

The foundation of the LW architecture is the addition of two new VRML nodes: ZONE and SHAREDOBJECT. These nodes extend several of the concepts of VRML to allow for shared worlds.

Zones Standard VRML has two high-level grouping mechanisms: worlds and scenes. VRML worlds are composed of linked scenes. A scene groups a set of related VRML objects. The objects in a scene must be geometrically bounded and a scene must allow for continuous navigation (i.e., no links need to be followed).

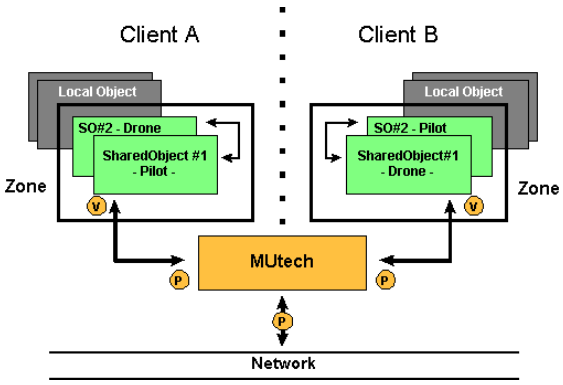

Figure 3.6 Living worlds architecture

LW extends these grouping mechanisms by allowing scenes to be further subdivided into ZONES. A ZONE is a contiguous portion of a scene that contains objects that are to be shared between clients. Zones can be contain standard VRML objects but they are primarily composed of SHAREDOBJECT's. (see Figure 3.6 on page 77)

LW uses Zones to distinguish areas of a scene that are to be shared. Zones define a range of control for a MUTech. A MUTech is associated with each zone and it is the responsibility of that MUTech to control the sharing of the objects in the Zone. Zones can be used by the MU Tech to optimize networking and handle area of interest management.

Shared Objects A shared object is any object whose state and behavior is shared across multiple clients. For an object to be distributed it has to be added as a child of a Zone. Once it is added, it is the responsibility of the containing Zone to facility object distribution.

Sharing an object in LW means keeping multiple instances of the object synchronized across multiple clients. This is done by distinguishing between object sources (Pilots) and object replicants (Drones). A shared object is composed of:

Pilot The pilot is an instance of a SharedObject whose state is distributed to the other nodes.

Drone A drone is an instance of a SharedObject that replicates the state of the pilot instance.

The Pilot of a shared object is the single instance that controls the true state of the object. There can only be one Pilot for each shared object and when the state of the pilot object changes,

those changes are distributed to all of the shared object's Drones. Remote nodes can use this Drone instance of the shared object to examine the current state.

When an object needs to communicate with a remote shared object it uses the local drone associated with that shared object. When an event is sent to a drone, the MU Tech routes the event over the network to the object's pilot. The pilot is then responsible for processing the event. This may involve updating the object's state or sending a new event to a local drone object. In either case the changed triggers another message that will be routed through the network to all affected shared objects.

MU Tech The MU Tech links the objects in the local scene graph to the external networking implementation responsible for distribution. It encapsulates the functionality for tracking and synchronizing shared objects. It is also responsible for distributing the state of shared objects and controls the access to the object behaviors. The MU Tech provides these abilities using VRML wrappers for the relevant node types. These features are added to the VRML system transparently so the specified virtual worlds to not have to explicitly handle distribution.

Example Consider the example shown in 3.6 — there are two shared objects: SharedObject#1 (SO1) and SharedObject#2 (SO2). SO1's pilot is on client A while SO2's pilot on client B. When SO1 needs to communicate with SO2, the SO1's Pilot object on client A sends a event to SO2's Drone object also on client A. SO2's Drone then routes this event through the MUTech to SO2's Pilot object on client B. SO2 then processes the event and updates any local state needed. Effectively as far as the object's on any client can tell, they are communicating directly with a "normal" object. It is the MU Tech and the associated Drone and Pilot code that takes care of all the networking details.

4 Research Overview

When we set out on this research, we look to previous work for inspiration. We found some interesting ideas, but there was no system that solved the core problem of provided a unified world model that could be fully distributed and allow for user extension. What we found instead was a number of interesting ideas that could be built upon to create a new type of software architecture for distributed systems.

4.1 Research Methodology

From the inception of the idea for this research, we realized that our ideas would only be relevant and proven if we could show that they could be realized in a software system. To many previous systems sound good in theory but failed to live up to expectations when used for real applications. Because of this we used used an exploratory research methodology validated through implementation. We investigated proposed solutions to the research issues through an iterative research process of design, implementation, and evaluation. The evaluation used test-case application scenarios designed to mirror real-world requirements. Based upon implementation and experimentation with these test cases, the design was iteratively refined to more completely address the project's research issues.

The test-case scenarios were key to this method because they provided a way to evaluate current progress. At any point we could use the test-cases to evaluate the capabilities that existed in the system and discover new issues that needed to be addressed. We believe that without these test cases there would have been no way to effectively evaluate and revise the system designs to ensure that they behaved correctly and successfully.

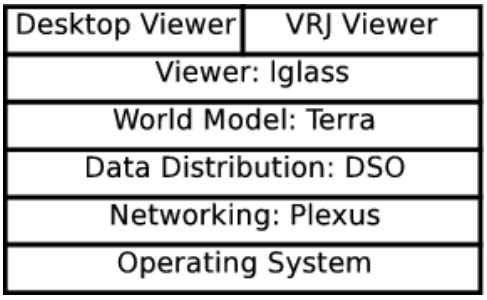

Figure 4.1 Subsystems and layers of Continuum

Development Methodology Throughout the design, implementation, and refinement we made a best effort to follow a standard software development methodology to manage the project. We did this to help keep the project focused and organized. Based upon our previous experience we decided to use the Extreme Programming (XP) development methodology [Bec01].

This methodology is built upon the ideas of continuous feedback and incremental development. This supported our research needs by being flexible enough to allow for experimentation and rapid changes in direction of core designs. It was lightweight enough that it did not get in the way but instead promoted creativity and exploration.

In retrospect, we do not believe this research would have been nearly as successful or progressed to it's current state without the use of XP development practices. The software ended up being so complex and requiring so many rapid changes in design and development that without a solid development practice being in place, the software would have taken much longer to develop and we would not have been able to try so many various options.

4.2 Architecture Overview

The architecture of Continuum is based upon a layered system. Each layer makes use of the layers below it to provide additional capabilities to users of the higher levels. By splitting the system into layers we were able to compartmentalize the complex array of capabilities needed for a CVE. These layers are shown in Figure 4.1 on page 80 and described below.

Operating System: At the lowest level, Continuum sits on top of an operating system ab-

straction layer that we have written called the VR Juggler Portable Runtime (VPR) [vpr06]. VPR has been developed as part of the VR Juggler project [vrj, Bie00, BJH+01], and it now serves as the basis for most cross-platform software written using VR Juggler. It is made up of platform-specific subsystems hidden behind cross-platform C++ interfaces. The subsystems can be changed at compile time allowing code to be moved from one platform to another without modification.

Networking Layer: Plexus — The Plexus networking layer provides the data communication and networking support needed throughout the system. It provides a multicasting message-based communication method using peer-to-peer communication.

Data Distribution: DSO — The DSO library provides a distributed shared memory abstraction based on distributed objects. This gives the other layers and applications a simple way to shared data with other nodes in the system.

World Model: Terra — The Terra library unifies the system with a common world model. This model is the basis for application data sharing, code composition, and resource sharing.

Viewer: lglass — The looking glass (lglass) layer brings the other layers together in viewer applications. Much as a web browser provides an interface to viewing and interacting with the world wide web, the lglass viewers gives users a way to interact within a CVE. We have implemented two viewers in lglass, one for the desktop and another for VR systems.

In the remainder of this document we will discuss the subsystem and libraries that make up these layers in much more detail. For each subsystem we will describe the conceptual model used to describe the capabilities provided, the software design used, the implementation details, and a brief discussion of the issues encountered and outcomes of the project as is pertains this this subsystem. After describing each subsystem we will close with a final section describing our conclusions.

5 Networking Layer

The networking layer of Continuum is called Plexus. Plexus is a network data routing library that operates at the application level on top of the TCP/IP stack. The focus of Plexus is to provide a common low-latency data networking layer for use within virtual environments. Plexus provides CVE applications with an abstraction that meets their common networking needs. By using a networking abstraction it allows developers to create higher level applications on top of Plexus while still allowing Plexus to evolve and improve in the future. This proved especially beneficial during our work because it let us to create a bare-bones version of Plexus initially that allowed us to continue working on the higher level tools without having to fully optimize and refine Plexus.

Plexus is not strictly limited to use with virtual reality (VR), but its design focuses on the needs of VR. Those needs include soft real-time responsiveness to enhance the users' suspension of disbelief.

5.1 Requirements

In CVE applications it is very common to communicate updates from one node to all the others in the shared environment, or at least to a subgroup of users that is interested in the update. For example, when one user picks up a virtual object and starts moving it, this update to the object's state needs to be sent to all the other users in this shared environment. This type of 1-to-N communication is an example of data multicasting. Multicasting network distribution is the process of sending a single data message to an interested group of listening entities on a network [1]. Because multicasting is a fundamental communication method in

[1] See multicasting on page 24.

CVEs, it is the primary communication model supported by Plexus.

Plexus provides multicast networking using a peer-to-peer distribution network. A peer-to-peer network is one where any party can initiate a communication session and connect to any other party. In a peer-to-peer network, all parties are equal in that they all run the same software that can act as both a sender and a receiver of information. In the case of Plexus, what this means is that every user in the CVE is running the same client software to communicate on the Plexus network. There are no centralized server nodes controlling the communication, and there is no need to connect to a central location. Instead a user simply connects to another user in the system and they are immediately a part of the entire Plexus network.

Plexus uses the peer-to-peer model because it provides several significant benefits to users, administrators, and developers. One of the primary benefits is network robustness and added fault tolerance. In a peer-to-peer network, there is no single point of failure. Any node can go down and the other nodes will continue to work as normal. By allowing every client to run the same software, it eases the development burden since there is only one code base to create and maintain. Peer-to-peer networking also makes software deployment easier because the same application code can be used for user clients, data servers, and hybrids in between these two extremes.

Plexus is targeted for low-latency communication, and hence, the goal of this project is to minimize the latency of distributed message passing. There are three main resource constraints on distributed message passing systems: CPU utilization, bandwidth, and latency. Available processing power is increasing constantly [Moo65]. Every few months, CPU speeds increase to give more processing power to use in applications. This does not imply that a software tool should not be optimized to make the best use of the resource but rather that there is no (known) hard limit. Bandwidth, similar to CPU resources, has been increasing exponentially, and it appears that this trend will continue for the foreseeable future. Latency, on the other hand, is a resource constraint that has a hard limit. Communication latency is limited by the speed of light. This is a hard constant that cannot be avoided. Moreover, the local area network type (e.g., Ethernet, Fast Ethernet, wireless), the backbone type (e.g., T1, T3, OC-12,

etc.), and the physical distance between sites contribute further latency.

Because latency is one of if not this most significant factor influencing the responsiveness of CVE systems, the design of Plexus focuses on reducing latency above all else. We work to achieve this by using soft real-time techniques and basing our design decisions and implementation choices on what will best reduce the latency of messages in the system. From the perspective of real-time systems, the deadline in the Plexus system is the time by which the network must deliver the message in order to avoid the side-effects of a high-latency system such as: jitter, distractions during collaboration, and loss of suspension of disbelief [2]. Since these effects are observed with differing severities with increasing latency, the system is a soft real-time system where the lower the latency of a message, the more valuable the message is to the system.

One way Plexus attempts to minimize latency is to allow the network to adapt dynamically to the current constraints so as to minimize latency in any way possible. Through dynamic network analysis, we Plexus attempts to determine where bottlenecks occur in the data flow. With that information, Plexus will try to find faster routes for the packets.

The routing used in Plexus does not try to replace the hardware routers and gateways that make up the Internet. Instead, it attempts to look at the flow of data at a higher level than the routers. Whereas the routers are concerned only with the next hop for a given packet, Plexus can look at the virtual connections that make up a Plexus network and make routing decisions based on that view.

5.2 Model

The model provided by Plexus combines these requirements with the domain specific needs of CVEs to provide a communication model with the following guarantees.

- There are a set of nodes N and,
- A set of possible multicast addresses M,

[2] See interactivity on page 31 for more details.

- $\forall n \in N$, n has a set of subscribed multicast addresses $M_n \subseteq M$,
- n can only send messages to a single address $d \in M$ at a time, and the message will only be received by the set of nodes $N_d$also subscribed to that multicast address where $N_d = \{\forall n \in N, \forall d \in M | d \in M_n\}$
- *Source ordering* [WZ99] is guaranteed for all messages delivery

Messages in plexus are addressed by providing the following information:

Field	Description
Type	multi-cast or direct
TTL	Allowed time to live measured in number of hops.
Domain	A GUID defining the primary domain that this message belongs to. This is the application or library that should care about this message.
Domain Group	A GUID that defines the sub-type for this message within the primary group.
Source	Address of the plexus node sending the message.
Contents	The data content of the message.

5.3 Design

A plexus network is made up of a set of nodes that share data by passing messages to each other (see 5.1). Each of these nodes is a hub into the shared Plexus network and has a unique address on the plexus network composed of a local IP address and port number. A single Plexus node is directly connected to some other number of Plexus nodes called FRIENDS. When a node sends a message or receives a message in the network it does it through its friends using the local node's software ROUTER. It is the router that handles the actual message passing for the local node and it is the router's responsibility to guarantee the delivery of messages to every node that is interested in them.

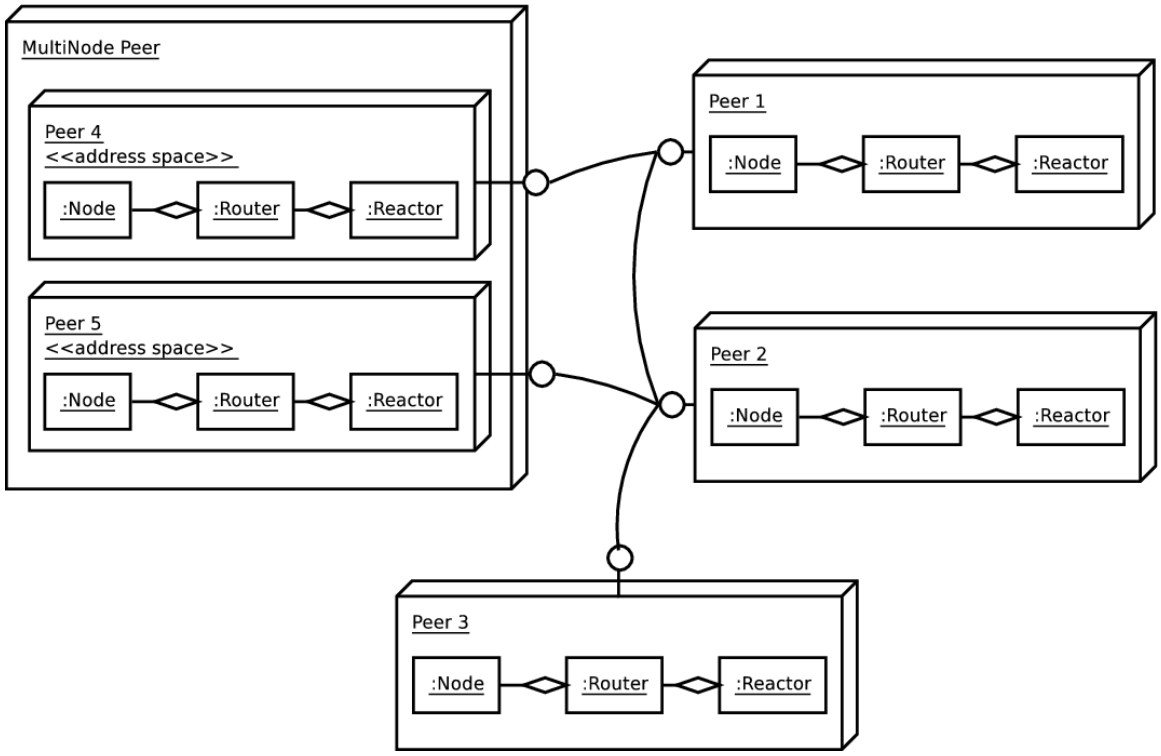

Figure 5.1 Network overview and deployment

Messages are delivered based on their multicast address. Once a message arrives at a valid destination node the local router delivers the message to any local MESSAGE HANDLERS that have been registered. Local applications and libraries receive messages by registering a custom message handler to intercept and process relevant messages. Message handlers can filter messages based on the message type and the destination address group.

An application that wants to communicate with a remote instance of itself creates a message, sets the message address to the multi-cast group the application is using, and finally asks the router to send the message. On the receiving side an application specific message handler must be registered to handle messages with that addressing and message type. It is then the responsibility of this message handler to pass the message on to the remote application.

5.3.1 Routing

To provide efficient methods for routing, Plexus allows pluggable routing algorithms that may be selected at run-time. These algorithms can exhibit dynamic routing behavior in and

of themselves through such techniques as network introspection and heuristic algorithms. By harnessing these dynamic techniques, we allow for the future development of powerful routing algorithms that minimize latency on the network to ensure that the users of the VE are not adversely affected by the introduction of network latency.

The current routing algorithms are all based on the concept of a "friend" relationship in the network. As described above, this type of relationship means that all nodes have a set of friend nodes that they communicate with directly. When a message needs to be sent to a node that is not a direct friend, the routing algorithms make use of the idea of sending the message to one or more friends and trusting that they will route it correctly to arrive at the correct destination. The process will repeat recursively until the message passing is complete.

This method of using friends to pass messages is commonly used in peer-to-peer networking. It is useful because it allows for simple algorithm implementations that provide a high degree of redundancy and reliability. Unfortunately it can also lead to using more communication and associated resources then may be strictly needed. For our needs this is not a significant limitation because we are primarily concerned with decreasing latency. In general friend networks do very well for decreasing latency and some dynamic algorithms can further improve the latency by taking advantage of small world behavior in friend networks [WS98].

In the future, more advanced routing algorithms could be developed to further improve performance. We are particularly interested in parent-child, clustering, and artificial life routing methods. A parent-child routing method could try to automatically organize the network nodes into a shallow hierarchy. If this hierarchy is based on areas of interest, latency, or network performance the it may be possible to use it to direct messages more optimally. Similarly a routing algorithm based on dynamic clustering could increase performance by trying to group communicating nodes into small tightly connected groups withing the larger network. The idea here is that if the groups are formed based on area of interest then messages can be routed quickly to only those nodes that have a direct interest. A final idea we have for dynamic routing is to use artificial life and evolutionary algorithm techniques to automati-

cally reorganize the network for more optimal performance.

We have experimented briefly with some of these techniques, but we have not created a fully working version. As a result of reviewing the current network literature we believe that ad hoc networking has much overlap with our work in this area and could prove to be a great source of ideas and inspiration in the future. We leave further experimentation in this area as future work.

5.4 Implementation

As described above, a Plexus network consists of a set of peer nodes that share data by sending messages to each other. The current implementation only allows one node per address-space per peer. The reason that there is one node per address space is that we do not have a standard way to access shared memory in a cross-platform way. In practice having a node per address space is not a significant issue because there is usually only one Plexus application per machine. In the case where there are multiple nodes on the same machine, Plexus will function correctly, but there will be more local communication and resource consumption than would strictly be needed.

User applications and high-level libraries interact with Plexus through the interfaces of the node and router classes. They can use these interfaces to configure the network, connect and disconnect from other nodes, send messages through the network, and query the status of the system. The messages can be normal Plexus messages or can be application specific messages. In either case, the interface to the routers is the same. The messages can be multicasted to a group of listeners on the network or they can be targeted directly at one destination node.

5.4.1 Router

As described previously, each node has an associated ROUTER. This router is responsible for passing messages on the network and for handing off the messages to local objects, called message handlers, that are interested in processing the received messages (see Figure 5.2 on page 89).

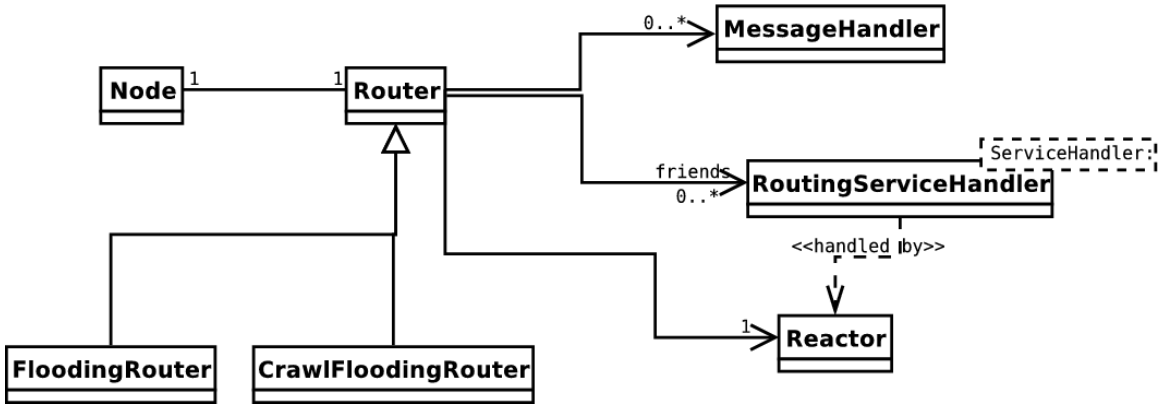

Figure 5.2 Plexus Node and Router classes.

The router makes use of a helper object called a REACTOR. It is the job of the reactor to efficiently multi-plex all the I/O processing tasks and to manage the network connections in the system. The reactor we implemented is modeled after the reactor pattern [SSRB00]. It is a single-threaded reactor that makes use of platform specific primitives[3] to minimize the processing overhead while simultaneously attempting to maximize the I/O throughput. We have found this design to work very well and it allows the asynchronous nature of the networking code to be encapsulated in a single location.

The core of the router is the implementation of the routing algorithm. In Plexus, the routers implement pluggable routing algorithms using a template method pattern [GHJV95]. There is a single common base class called *plx::Router* that presents a common interface and implementation of the base router functionality. This class makes use of several internal template methods that are implemented within the actual router implementation that is derived from the router class (see FloodingRouter in the Figure 5.2 on page 89).

5.4.1.1 Flooding Router Algorithm

The first routing algorithm implemented for Plexus was a basic flooding algorithm [Wit01]. Each message is sent to each Node in the network and if that node has not yet seen the message it forwards it on to its friends as well. This routing method results in a rapid flooding of the network. In the worst-case this can result in a significant amount of un-needed messages

[3]The primitives used are I/O notification methods such as the POSIX *select()* call.

Algorithm 1 Flooding router algorithm

```
for(each message)
if(haven't seen message before)
{
   Send message to all peers Pass message to message handlers
}
```

being sent, but in practice the algorithm seems to work very well.

We chose to implement this algorithm first because it is a relatively simple algorithm that has a straightforward implementation. Ease of implementation was important because it allowed us to have a base algorithm with which to validate the rest of the system. The flooding algorithm also has the added benefit of routing message with very low latency in small networks.

This router is used as a baseline for comparison throughout the system. Because it works well on small networks it has also served as a good test-bed router allowing for the development and use of higher-level libraries and applications. Being able to build upon the Plexus library while the system is still in development has been very valuable. We have been able to refine Plexus continually while continuing to develop other software that makes use of the library.

5.4.1.2 Crawl Flooding Router

The CRAWL FLOODING ROUTER (CFR) was our first attempt at an adaptive router. It is based upon the flooding router but has some added abilities that allow it to adapt to the network at run-time by connecting to new friends or disconnecting from existing friends based upon the currently observed network characteristics. This adaptation is guided by maintaining a *potential peers* list containing a list of peers that are currently under consideration for connection. The potential peers list maintains a set of metrics for each peer that is currently being considered.

These metrics include:

- Ping time: The turn-around time of a ping packet to the node.
- Load information: A set of metrics that reflect how loaded the peer is currently.
- Throughput: A set of metrics that approximates the current data throughput.
- Reaction time: A metric that estimates the amount of time it takes to route a message through the node.

The selection and connection policies are based upon these metrics. The router uses the metrics to decide whether to maintain its current connections, add a new connection, or close a connection.

Updates to the statistics for the peers occur through a variety of methods. When a node first connects to a new peer it announces itself to all its new second-level peers (peers of the new peer). If these peers did not previously know about the new node, then they ask the node for its current status. Disconnection works much the same way except it is the responsibility of the peer that the node disconnects from to tell its peers about the disconnection. When a node gets a notification that another node is disconnecting, it removes the departing node from its potential peers list.

The CFR also supports updates during execution through periodic updates where a node tells its peers of its current state. This can either be fully periodic or can include a policy that only sends the update out when there has been a significant change in the node's state. The definition of what is considered significant is configurable. It is also possible for a node to request an update from one of their peers. This can be important for cases where the node has either lost the data for a peer or wants to make sure that it has the most up to date information before it makes a topological change to the network. In all of these cases, the information policy is fully configurable and is able to change at run-time.

Because such a large number of parameters are collected and the parameters have very complex relationships to each other and the current system performance it is very difficult to write algorithms to make good choices based on these values. In the future, we hope to use genetic algorithms to evolve methods that aggregate these metrics into an algorithm

that optimizes the network performance. For now, we have implemented a simple heuristic method that is based upon the ping time metric. The idea behind it is to try to find "close" nodes on the network and connect to them. The current algorithm does not take advantage of disconnecting from peers, as we have not had time to fully explore and debug this capability.

5.4.2 Message Handlers

Routers not only send messages to other nodes, they also pass the message to message handlers that have registered with the router as being interested in processing the messages. All message handlers are derived from a common base class that defines the interface that all message handlers must support. The interface consists of two methods: SHOULDHANDLE() and HANDLEMESSAGE(). The router calls the SHOULDHANDLE() method to determine whether a message is of interest and if it is, then it passes the message to the HANDLEMESSAGE().

We are working on a new interface method that allows the handlers to enumerate all possible message types that they may be interested in. This is needed to allow routers to communicate information about what messages should be sent to a given node. It will be needed by more advanced routing algorithms currently under development.

5.4.3 Reactor

The routers make use of a REACTOR to manage the actual network traffic (see above diagram). The reactor is an implementation of the reactor pattern [SSRB00] which is used to demultiplex and dispatch events that are delivered to the router. In this case, the events are network events such as connection attempts and data ready notifications.

The Plexus reactor works by maintaining a list of the current sockets connected that can serve as event sources (ServiceHandlers in the above diagram). These socket handles are encapsulated in event handler objects that not only contain the handle identifying the potential event source, but also the event processing code to execute when an event is received. This allows the event handlers to encapsulate completely the communication protocols and mes-

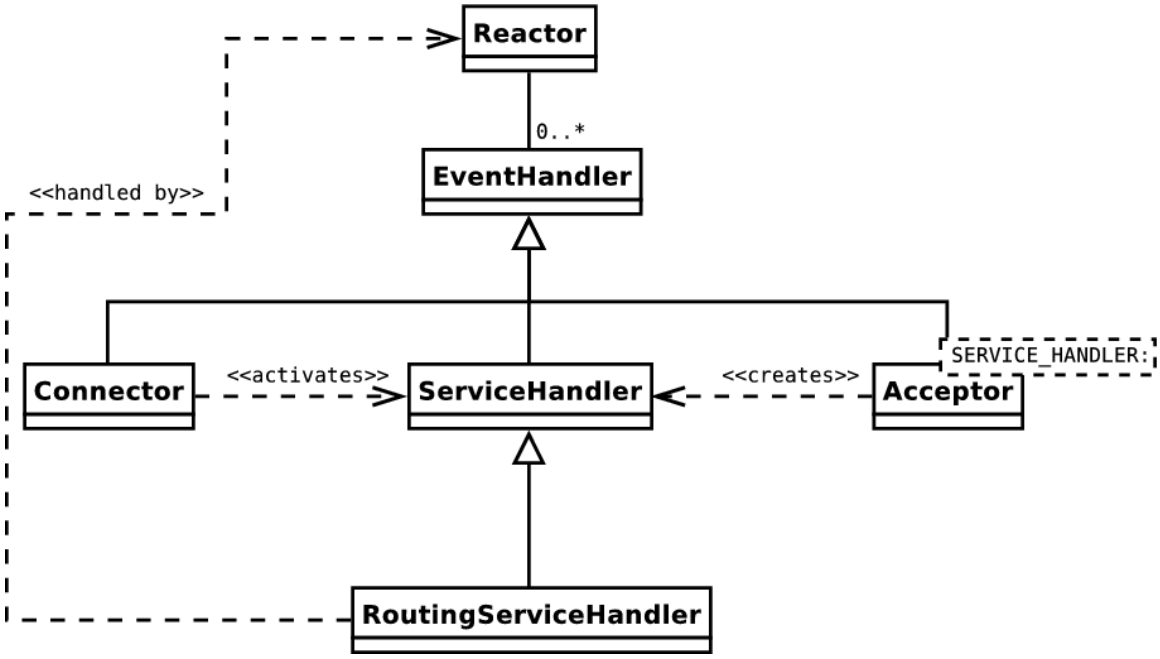

Figure 5.3 plx::Reactor classes

sage handling in a single place without having to worry about the details of the actual socket handling.

The reactor design leads to a system that is high-performance, flexible, and modular. Performance is optimized because the application never has to block or wait for a network connection since it only processes a connection when it is guaranteed to have an event. The system can optimize throughput because it can always be processing whatever connections have data at a given time.

This design leads to modularity by separating application level event processing from the actual dispatching method used. It also allows common event handling components such as connectors, acceptors, and service handlers to be implemented. The reactor captures the details of multi-threading and synchronization in a single place thereby simplifying the rest of the code in the system. This allows for the replacement of the reactor with more advanced demultiplexing implementations without requiring changes to the rest of the system. Statistics Collection

From the beginning of the project, Plexus has been designed to allow for the collection of

performance metrics and statistics about how the network is performing. The routers, reactors, and service handlers in Plexus are instrumented to collect this information at run-time. Routers collection statistics about the messages processed, messages dropped, and messages routed. Reactors collect statistics about that duration of even selection and event servicing. The service handlers collect information about bandwidth consumption and message latency.

Plexus includes methods to collect this information as basic counts, counts per unit of time, and continuous rates over time.

5.5 Discussion

In the process of developing Plexus we considered many various options. We weighed our choices against the needs of CVE systems, what was possible with current software, and the difficulty of the associated implementation and debugging. Our focus was to create the best platform we could for supporting the development of CVE applications.

Our previous systems had always been based on client-server based architectures. These systems worked very well and resulted in easy to understand code but they proved to be too fragile and limited scalability. Based upon this experience we decided to pursue a peer-to-peer based architecture with Plexus. Although this made the resulting code more difficult to develop and debug it has allowed us to create a better overall system. Plexus is now able to work reliably and robustly in a variety of conditions without a central point of failure. We also think that this network architecture has lead to improvements in the end-user applications.

Initially we put a great deal of thought into the type of addressing modes to support. We knew that group addressing through multi-casting was needed but we did not know if other methods would be needed or how much use the various methods would see in applications. In the end we found that multi-cast communication was the most important method to support. We added support for direct message passing, but we did so on top of the multi-cast communication method. By do this we were able to leverage the existing codebase and minimize the additional effort needed.

After examining the various design alternatives, we arrived at a solution that supported

our needs and that appeared feasible to implement. This design could still be further extended and refined in the future to provide more complete functionality and better performance.

6 Data Distribution

A CVE consists of a large amount of state information that needs to be shared among all participants of the system. For example the position of an object in the world, its current color, or information about other users in the environment. All of these are examples of shared state about an object, or more generally, a shared entity in the CVE. The core of any CVE is a large shared repository of entities and their associated state. This shared collection fully defines the entirety of the the shared environment.

Thus one of the most fundamental needs in a CVE is to manage this shared state. It is the responsibility of the CVE software to provide access to this state information and keep the data in sync across all participating nodes so all users see a consistent and coherent view of the virtual environment. If the system fails to keep this state consistent, then at best users will begin to loose the feeling of being in a shared space and at worst applications based upon this system will fail to execute correctly.

While it would be possible for a user or other layers of Continuum to use the message passing support of Plexus directly to keep this state information consistent, we have found this to be overly burdensome on the developers and users. Instead we provide another layer on top of Plexus, called DSO.

DSO is a system for managing distributed shared objects and providing a single consistent interface to developers and users of the Continuum system. The DSO subsystem has many responsibilities, but the most fundamental of these are: holding entity state information, distributing updates to all nodes in the CVE, controlling access to entities, and providing data persistence. We have found that by centralizing all of these capabilities into one subsystem we derive several related benefits.

First, it succinctly captures a great deal of the most complex aspects of the implementation behind a relatively simple facade. Encapsulating the implementation in this way allows the developers to refine, test, and expand the capabilities of DSO in isolation from the other parts of the system. The benefits of maintaining this separation of concerns during development and maintenance have been critical to our development. It is our experience that this has dramatically increased the speed of development by allowing us to create an extensive automated testing suite which lets us continually refactor the code as needed and still maintain confidence that the system keeps working successfully.

A second benefit we have seen is that using DSO seems to decrease the complexity of the system as a whole. Developers do not have to constantly worry about the myriad of potential issues involved in using a distributed system. Instead they can focus on writing their code using data objects that appear to be relatively standard. DSO will take care of the intricate details of synchronization and distribution for them behind the scenes.

This last benefit translates directly to application developers as well. When a user writes a collaborative application using Continuum they can make use of a concise object model for all their data needs. They can rely upon this data working in a distributed manner with very little worry on their part, thus simplifying application development dramatically.

Because DSO was created to support Continuum, it was designed to specifically target the unique needs of CVEs. Although we have based some of our ideas on previous work creating shared memory libraries , we do not provide support for some of the more general mechanisms found in these systems. Instead we actively take advantage of the domain specific behavior and needs of CVEs. This has allowed us to provide additional capabilities and performance enhancements because we know how the library will be used and what constraints will be most limiting.

6.1 Design

There are two main components in the design of DSO. There is the object model used for storing shared data and shared memory subsystem responsible for exchanging messages with

other nodes to make sure everything remains synchronized. In this section we will discuss the design of each of those components and how they fit together to form the DSO subsystem.

6.1.1 Object Model

We know from previous work that it is critical for the object model of DSO to provide extensive support for run-time addition of types. A CVE system will not know ahead of time what types a user in the collaborative space may want to use or what objects it will need to handle. In many cases users may introduce a new object of a type that is completely unknown to many of the nodes until it was added to the environment. When a new type is encountered the system must be able to proceed without interruption and handle this new type transparently. Although some systems try to add this support by dynamically loading new binary code at run-time we have found that this does not scale and can lead to a very fragile system. Instead we focused on designing a system with run-time extension capabilities in mind from the beginning to allow the types to be added and successfully handled at any time during execution.

Providing object introspection and type reflection capabilities is equally as important for a CVE. Because new types are being continually added to the system, the entire subsystem must handle objects using introspective interfaces so it can work with any objects in the system. For example the system may not always know the full type information of a shared object and it should not be required to know this information to simply process the messages needed to exchange object information. By providing a reflective interface, DSO and applications written using it can automatically adapt to whatever objects are in the system and use introspection to access the properties of the objects without having pre-existing knowledge of an object's type.

To provide these needs, DSO stores data using a property based object model [Rei02, FY98, vR06, Fow97]. The primary concepts that make up the DSO object model can be seen in Figure 6.1 on page 99. The technical realization is a bit more complex and is shown in Figure 6.2 on page 99.

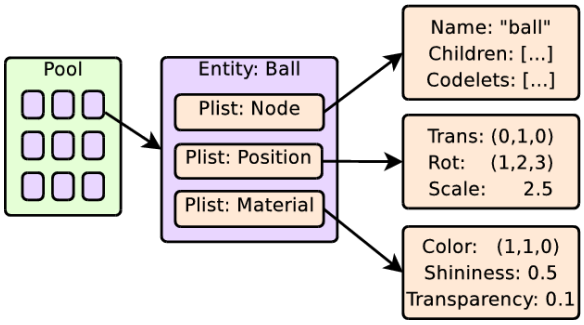

Figure 6.1 Concepts in the DSO Object Model

As can be seen in the diagram, the fundamental type in this model is the PROPERTY. A Property contains the data for a single piece of tracked information in the system. For example a property may define the color of an object, it's position in the scene, or some state information associated with that object in a simulation. Multiple related properties are held within a PROPERTYLIST. Properties in a property list are looked up either by name or by index.

Both Properties and PropertyLists have associated type information. In the case of a property, the type information is held in a PROPERTYTYPE. A property type specifies the data type for the property, the name of the property, the cardinality for the property. The allowed data types are limited to a set of fundamental data types and two extended types specific to this type system. The full list of data types include:

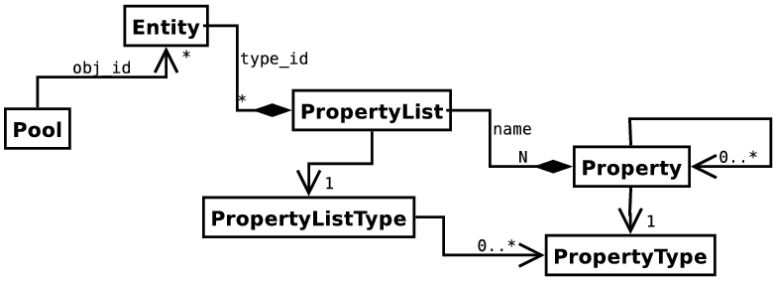

Figure 6.2 Class details of the DSO Object Model

Type	Description
Uint8/16/32/64	Unsigned integer types of fixed bit length 8, 16, 32, and 64 respectively.
Int8/16/32/64	Signed integer types of fixed bit length 8, 16, 32, and 64 respectively.
Float	A value that can be stored in a standard C float type.
Double	A value that can be stored in a standard C double type.
String	A string of arbitrary length.
Array	An array of properties.
Value	An embedded property list value.
Ref	A reference to another DSO entity.

The details of the the properties held by a PropertyList are defined by the the PROPERTYLISTTYPE. This type defines the properties to allocate for a new PropertyList, their associated types, and what names to use to refer to the property within the list. The PropertyListType corresponds to a class type in an object oriented language and as such, this is the point in the object model where users can introduce new types to extend the system. A user can define a new type in DSO by creating a PropertyListType with a unique type id and a list of property types to use when allocating PropertyLists of that type.

The next level of grouping in the DSO object model is the ENTITY. An Entity provide a holder for multiple related PropertyLists. For example a ball in a virtual environment may have a PropertyList that defines its physical position as well as one that holds information about its color. Each of these is an independent PropertyList, but they are held together in a single Entity that defines the full state of the ball. Entities can wrap any number of PropertyLists of differing types. Users may look up a specific PropertyList held in an entity by using the type id of the desired PropertyList.

Because all objects in the system must be distributed and tracked, there is a single point of management for all DSO objects. This point of entry is the object pool. The object pool holds

all of the Entities in the system. The contained entities each have a unique id that can be used to look up the entity. When a new entity is created it is automatically added to the Pool.

6.1.1.1 Type Management

One of the goals for DSO was to make sure the type system was flexible enough to represent nearly any data structure while at the same time providing reflective interfaces and generative tools to help support software developers. We went through several iterations of the design before we were able to meet all these needs, but the final design provides these capabilities and more.

As shown in Figure 6.1 on page 99 there is a PropertyListType that defines the structure of a uniquely identified property list structure. This object holds a list of PropertyType objects that each fully define the details of a single property in the property list. These two classes fully define the type system for the DSO system and at provide a reflective interface for run-time introspection. Given any Entity or PropertyList at run-time, code can query the PropertyListType or contained PropertyListTypes. This object provides a common interface to get the type id, the name of the type, and a list of contained PropertyType objects. For each of these objects the code can discover the name, type, and method of access for each individual property.

This reflective interface is key to much of the code in DSO and at higher levels because it allows us to write algorithms that will operate on any objects in the system even if their types are not yet known. The serialization code in DSO uses this code to write and read binary and text representations of Entities, PropertyLists, and Properties on disk and on a network. We have created development tools that use these interfaces to allow the user to see and edit in-memory data structures at run-time. One very powerful use of these interfaces was in creating the python bindings for DSO. Python is a fully dynamic programming language. It allows for run-time creation, modification, and removal of types, methods, variables, and every other part of the run-time system. When we created the Python interface to DSO we took advantage of these capabilities and combined them with the reflective interfaces of DSO to dynamically

create run-time variables and types to interface with DSO. For example the Python interface provides an type tree variable that automatically updates itself based upon all the known types currently loaded into DSO. This dramatically simplifies the user experience with using the Python interfaces and allows for the use of run-time types that are not known before execution.

It can not be overstated how important the reflective interfaces are to the functionality of DSO. Simply stated, without the reflective interfaces, DSO would not be able to function.

Although it is possible for a user to create a new data type (PropertyListType and associated PropertyTypes) at run-time, doing so has some significant limitations. Any other user that wants to use that type must replicate the type creation code by using the same unique type id and property specifications. The code to create these types is complex, redundant, and fairly fragile because it has to change if the underlying type system is upgraded or modified in any way. We realized very early in the process that it would be better to create a type system that could be entirely data driven. As a result, all types in DSO can be specified using XML.

All type specifications are stored using XML files that can be loaded, saved, and processed at run-time. This allows the core object model to remain very small but still provide support for very complex relationships and data structures at run-time. The structure of the XML type files must conform to a predefined XML schema. This schema allows users to define new property list types, contained properties, and to reference other types as both base types and and embedded property types. An added benefit of using XML files and an XML schema for type definition is that we can make use of the plethora of XML tools available for loading, processing, and editing the data files.

Because we have all of the meta-information about the types available in XML files we can also create tools to assist developers with these types. One of the areas we have used this information is for generating C++ headers to assist with accessing the types at run-time. When we started using the DSO system we quickly realized that there was a need to have access to this meta-information in user code. For example to allocation PropertyList of a

given type, the user needs to pass that types unique id to the system. Although it is possible for the user to put this id in their code everywhere they need it, this is a pain for developers and is error prone. The same is true when accessing properties of a property list. Although the user can access them using their string names, this has a performance cost associated with it because of the string comparisons and hashes that have to occur behind the scenes. In many cases it would be more optimal to simply access the property based on an integer index within the PropertyList. Once again, the user could use manually code this index in any place they need it, but this requires them to know what each index is for each property. It was obvious to us that these were areas that could use significant improvement. The solution we arrived at was to create a code generation tool that uses the meta-information available in the XML type files to generate C++ header files with symbols for each of the identifiers the user may need at run-time. This way the user could use a symbolic name for the type ids, type names, property indices, and property names they need to pass to methods at run-time. The code generator would ensure that the correct values are used and the compiler will make sure the user is using valid names at each location. This solution increased the ease of use for the developer and dramatically decreased the number of errors related to using incorrect values.

Although we have primarily focused on using type meta-information for code generation tools there are other tools that could be created. It would also be possible to create other tools to assist developers such as type dictionaries or even automated data mapping tools. This is an area we see for future expansion and research.

6.1.2 Shared Memory

The shared memory system in DSO is the heart of the CVE. This system must distribute all data changes in the system and ensure consistency of data to all users. Every other part of the CVE architecture is built upon this foundation and relies upon the service it provides. Because this system is so critical, we put significant effort into refining the design of the system to best meet the needs of a CVE system. Where possible we have optimized the performance and

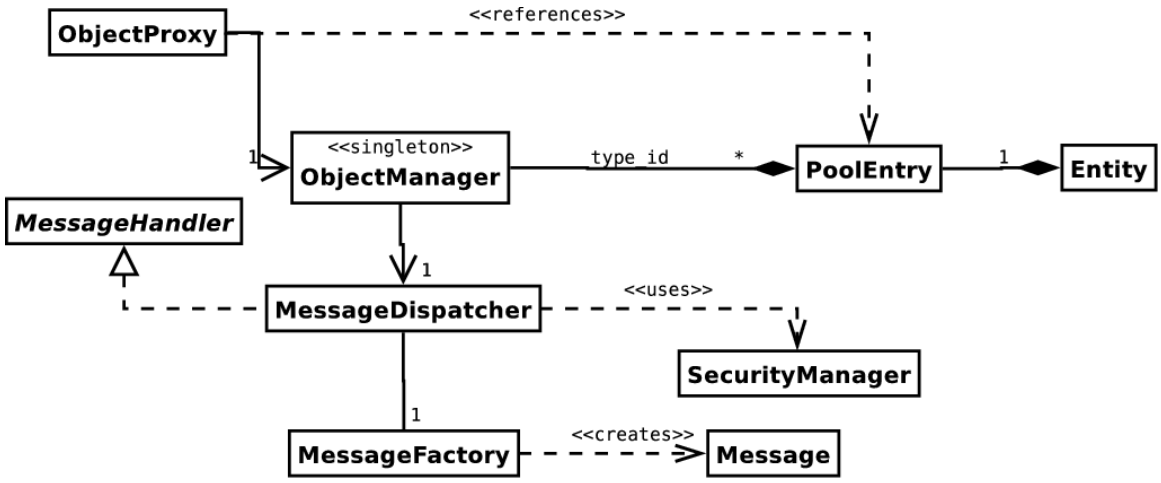

Figure 6.3 DSO Architecture

usability to take into account the unique characteristics and usage patterns of a CVE system. In this section we will describe the key components of the system and the insights found while refining its architecture.

6.1.3 Core

The core of the shared memory architecture is to provide a object pool where users can allocate new objects, update existing objects, and rely upon these changes being distributed to other remote users in a consistent and timely manner. The goal is to make the interface for supporting these use cases simple and consistent while hiding the complexities of the implementation.

The architecture builds upon the base framework provided by the object model described in the previous section. Figure 6.3 on page 104 shows the external and internal details of our design and how it relates to the object model. The interface for a user consists of two key abstractions: Entities and ObjectProxies. These abstractions provide all the capabilities that a user or library needs to shared data throughout the system.

Entities As in the object model, Entities are the user accessible data objects. Distributed Entities are wrapped by a POOLENTRY class and are held in an object pool under the control of an ObjectManager.

The PoolEntry wrapper extends the base abilities of an Entity with the capabilities needed to support use for shared memory. In DSO we need a way to simultaneously allow reading and writing of multiple buffered versions of an entity (see 6.1.3.4). PoolEntry encapsulates this feature and provides the interfaces needed for using locks to synchronize the access to the multiple buffers. When the system needs to serialize or deserialize the buffers it uses code from PoolEntry to do this as well.

It is the ObjectManager's responsibility to keep track of all objects in the CVE and control all access to them. The ObjectManager makes use of a MessageDispatcher which controls all message handling on the network. The MessageDispatcher is a realization of a Plexus MessageHandler and thus uses a plexus network for all of its communication needs. The MessageDispatcher in turn uses a MessageFactory for building messages from network packets and a SecurityManager for encrypting and verifying the messages it handles. This entire object collaboration is responsible for maintaining consistency and coherency of the data throughout the network.

One seemingly small feature of entities that has profound impacts is they are identified using a UUID. By using a UUID we were able to solve the difficult problem of how to allow multiple users on multiple nodes to simultaneously allocate new objects without requiring a single central arbiter to provision ids. The use of a UUID simplifies this problem by allowing entities to be uniquely identified and allocated throughout the entire shared network. The user of any node can allocate a new object and be assured that the new object has a unique id throughout the network.

6.1.3.1 Object Proxies

Because of management needed in the system, we found that we could not give the user direct access to the entity and still maintain the ease of use required. Instead the design uses a proxy pattern to wrap all access to entities behind an interface provided by the OBJECTPROXY class. This ObjectProxy class is the user accessible interface to all data in DSO. Using an object proxy a user can use pass through methods that allow them to allocate new data, lock/unlock

objects, update existing data, and detect changes made by other users on remote nodes. All of these capabilities are actually provided by other objects that make up the run-time collaboration of the system, but to the user there is a single interface that hides the details of the specific implementation. From the user's perspective they simply modify a property on one node and then change shows up on all remote nodes. Behind the scene the proxy makes the correct calls to the ObjectManager and PoolEntry to trigger the needed steps in distributing and coordinating the data within the system.

6.1.3.2 Change Detection

When using a shared memory system it is often important to detect data changes and respond to them. For example if a remote node changes the color of an object in the scene, then the local view must detect this change and update the state information used to render that object locally. The design of DSO supports change tracking at multiple levels. First, the ObjectManager maintains a list of ids of all entities that have changed in the previous frame. Second, the entities and property lists themselves maintain tracking information about which contained items have changed. User code can follow this chain of change information to detect exactly which pieces of state in the system have changed.

User code can monitor changes by either checking the list of changed entities each frame or registering a callback with the ObjectManager to be called any time it detects a change to an entity. In either of these methods the user can select whatever level of granularity they need for the task at hand. If they only need to know that a property list has changed then they can monitor at that level using the entity change tracking. If on the other hand they need to know only if a specific sub property within a property list changes, then they can monitor at that level instead. For example the system may say that the entity representing a ball has changed, then the user code can look at that entity and programmatically determine which contained property list has changed and further which property within the property list changed. We have found that this system of change tracking allows the flexibility we need to support a wide variety of update and monitoring methods.

6.1.3.3 Locking

As with any distributed system we have to provide a way to coordinate access to a single entity from multiple nodes. For example a user on two different nodes may both want to change the position of an object in the scene. When this happens, we need a way to coordinate the two nodes so the changes maintain data consistency (both temporally and absolute). In DSO we have chosen to solve this by providing a token based lock for every entity.

At all times there is one node in the system that holds the token for a given entity. Whenever a node wants to make changes to an entity it must request this token before it can begin making changes. As part of this token exchange we ensure that the receiver has the most recent version of the entity's data. Updates to state are only allowed from the node with the token. If another node attempts to send an update then that update is ignored. When a node releases the lock, then it automatically sends the token to any other nodes that are waiting for it. If there are no other nodes waiting, then the node keeps the lock until it receives a new request.

At first we thought this method may be a little too brute force to provide the interactivity we need, but it has proven to work well in practice. It often ends up that a single node makes most of the updates to an entity and in this case that node usually has the token for that node. This can be seen as an extension of the principle of locality to distributed systems. A node that references a given entity often references it again in the near future. Because their is a token per entity we are able to take advantage of this locality to increase the overall performance of the system.

One additional capability we would like to add in the future is for a node without the token to send an update request to the node with the token. If the update it acceptable, then the token holder could modify their copy of the entity and send out the update. We believe that this would further decrease latency in the system and prevent some forms of trashing that can occur if two or more nodes are both attempting to sequentially modify an entity.

6.1.3.4 Buffering

While designing this architecture we encountered several interesting insights that allowed us to tailor the architecture and implementation to the needs of a CVE. One of the first things we noted is that we could use data buffering to take advantage of the fact that CVE software if frame-based. When a user changes data in the system they do not need or want it to show up as a change until the next time through the frame loop. This means that there is an inherent lag between the time a change takes place and the time that it needs to show up. We can use this lag to our advantage to mask the network latency in the system. To do this we added multi-buffering to all entities. When the user modifies an entity they are modifying one copy of the entity and when they read data from an entity they are using another copy of the entity. This allows us to asynchronously receive updates from other nodes and send updates out to other nodes mid-frame. As described above, this buffering is hidden from the user through the use of the PoolEntry and ObjectProxy classes.

All entities maintain three buffered copies of themselves:

Stable: This buffer holds the version of the data that is stable throughout the frame and represents the current distributed state.

Working: The buffer that holds any modifications made locally or modifications read from remote nodes.

Working_Copy: This buffer is a copy of the most recent consistent snapshot of the Working buffer. It is used to provide a performance optimization to make sure that we can always grab a copy of the most recent completed updates and move them into the stable buffer.

To ensure that only one process is writing to a buffer at a time we use locking on the buffers. This locking extends the token based locking described above and manages all access to the entity buffers. The process of locking the buffer selects which buffer the application will read and write from when it accesses the entity. When the application unlocks the buffer, the system can then look for changes and if there are any, it can begin to process them by sending the updates to other nodes and locally marking the changes for the change detection system.

This same locking method provides a simple solution to allowing multiple local threads to access the entities. The design says that in order to write to an entity the local thread must hold the lock for that entity. Any writes to an entity without the lock held are considered invalid. We solve the problem of multiple concurrent threads by making the locking thread acquire a shared mutex. Only one thread can hold the mutex at a time, so this serializes all access from multiple local threads.

6.1.4 Dispatcher

The message dispatcher forms the backbone of the DSO system. It has the responsibility of processing messages from remote nodes and sending message on behalf of the local system. The methods within the MessageDispatcher implement all the distributed algorithms in the system.

6.1.5 Security

In addition to the to primary requirements of a object model and a shared memory system, a deployed system needs to provide support for security. Although a detailed treatment of security is outside the scope of this work, we have provided an initial security subsystem that could form the basis of future work.

The security model provided by this subsystem is based on the following observations:

- We need to ensure that only objects that are authorized to change an entity are allowed to do so.
- We need to ensure the authenticity of any notification of a data change.
- We need to ensure messages passed through the peer-to-peer system can not be intercepted and modified for later replayed.
- More generally, we need to guarantee that messages in the peer-to-peer system are authentically coming from their identified source.

For the purposes of this work we choose to implement a rather simplistic security system based on a public key infrastructure (PKI) using the Crypto++ library[1]. This system is based on the idea that each user will have a unique login to the system and this login will have an associated public-key/private-key pair. Further given the user id, any node in the system can easily find the associated public key. The PKI pair is used throughout the system to sign and encrypt messages and data packets. To simplify the system, we use the user's public key as their user id within the system. Because of this, we need a key with a short length because the key will be part of most messages sent by the system. We found that elliptic curve cryptography provides a good balance between security and key length [Mil85, Kob87]. For our current system we choose a key length of 112 bits. For purposes of comparison, this key length provides approximately the same security level as a DSA/RSA key with a length of 2048 bits [Sta00].

All security within the system is handled at the message level by signing every message sent by a given node. This allows other nodes to verify that the data did in fact come from the source node. When a message is received by a node, the signature is checked and if it is invalid, the message is immediately dropped from the system. The system prevents unauthorized data changes because the token handling in DSO only allows updates to come from authorized nodes. If the data change comes from an un-authorized sources, then the update is ignores. Replay is prevented by a sequential counter that is an inherent part of the existing routing algorithms. If an old message is sent on the network it will not be routed within the network.

We realize that the current security subsystem is not exhaustive and needs further refinement. It has served our needs but it needs much additional work. In future work we would recommend re-examining the security subsystem. In particular we need to find a better solutions to managing the user id and PKI system. We believe future work could draw inspiration from efforts such as OpenID that create distributed user id infrastructures [ope].

[1] http://www.cryptopp.com/

6.2 Discussion

Because all data in the CVE is based on using the distributed objects of DSO, it was critical to get the design of this system right. A bad design decision here would have a ripple effect on the rest of the system that would at best make higher level designs awkward and at worst make the system entirely unstable and unworkable. During the design and implementation phases of DSO we encountered many things that went well and a few things that did go as well.

One area that did not go well was the development and debugging of the distributed algorithms. Building working distributed object systems is hard. What seems straight forward on paper is very difficult to implement due to all the corner cases and the inherent complexities of distributed software systems. Debugging DSO ended up being a significant undertaking. During the initial design process we decided that the only way to really ensure quality code would be to use extensive unit testing. These test worked well, but we ran into problems where the test suite could not fully replicate all the issues we would run into with a fully deployed system. In the end we had to re-write the unit test harness several times before we finally found a setup that worked reasonably well and showing bugs and oversights in the DSO back-end.

What we finally settled on was a unit test framework that created multiple separate processes. Each process executed in it's own memory space with it's own copy of the DSO system. On start-up each process received a unique identifier that allowed it to select the test code path to execute. We also provided basic inter-process synchronization primitives using the underlying plexus network. This allowed us to write code that would for example stop at a barrier waiting for all processes. Although this setup was more complex then our original test suites, we were able to provide APIs and macros that still allow the tests to be written in a straightforward way.

An area that went very well was the design of the interface. It is possible to get a great deal of capabilities with very simple rules and requirements. We purposely create a non-distributed version of the property list code first so we could explore what a minimal interface

would look like. only after we whittled this interface down to a bare minimum did we begin to morph it into a distribute object system. This worked well because it made us focus very early on the developer experience and gain an understanding of what exactly the API should provide.

Because the external user interface is entirely captured in the Entity and ObjectProxy classes, the implementation details are completely encapsulated and separated from the users. This is a key advantage of this architecture because it provides a great deal of flexibility in implementation. As long as the Entity and ObjectProxy behave as expected we can vary, refine, and optimize the implementation as needed. When we implemented this design we took full advantage of this fact by initially creating a brute force implementation that worked and then later refactoring it to take advantage of further optimizations and refinements.

During development we investigated several alternative designs before we arrived at our final system. One interesting alternative we looked into was a to base the system on the concept of tuplespaces [Gel85]. In the end we opted for a more direct approach that feels more like standard object access. It would be interesting though to see if there was a way to generalize this using tuplespace concepts to create something with more mathematical rigor.

The final outcome of this work was a working shared object system that satisfied all of our original requirements. It could be improved upon but it provides what we need for the higher layers of the system.

7 World Model

While Plexus and DSO provide a way to communicate between nodes and share data within a distributed application, they are still very low-level. From our previous work we know that we need something higher-level to hide these low-level details from developers and instead let them focus on creating engaging and meaningful CVEs.

One of the goals of this research is to design a software architecture that can provide this higher level representation in a way that allows for distributed composition of the resulting CVE. We call the layer that provides this representation Terra. Terra provides a common world model that allows for the aggregation of code, behavior, and data into one single shared structure that is user extensible. The world model forms the backbone of the shared environment. Terra manages this model and uses DSO and in turn Plexus to handle the low-level details of the system.

We need Terra because it allows us to bring together the code, behavior, and data into one single structure. When a user wants to add behavior to the system they simply add it to this structure. When state changes in an entity, it is made in this shared structure and then all the remote nodes see the update. By capturing the entire world in a single structure that represents both code and state we provide a relatively simple conceptual model that allows for a great deal of flexibility in development.

The unified world model allows developers and users to focus on using the shared virtual world at a higher level. Instead of thinking about network messages, data updates, and distributed algorithms, they can focus on the behavior code and entity data. The advantages of this separation can not be over emphasized as it is the key to raising the bar and allowing developers to create new systems without worrying about the complexities of the underlying

details.

Because the world model is shared across the entire set of collaborating nodes, it allows for world composition. Any user on any node can add entities and behavior to the scene. In turn, the code for this behavior can be running on any node in the system. There is no need to have a single centralized system or to restrict the extensibility of the system. This puts every user on the same footing and allows for complete freedom in extending the world.

Another benefit of Terra is that it provides a central point of control for distributed resources. All resources in Terra are represented by unique identifiers. Any node in a CVE can use one of these identifiers to request a copy of the resource and Terra will automatically find a copy of that resource and bring it local to the node for use. This can be used for example to propagate code across the system, find data files needed for a scene, or load any other resources that is needed in the system. The developer does not have to worry about manually downloading data or sharing it with other users. They only add the resource to their local system and rely upon Terra to distribute the data where it is needed when it is needed.

In many previous systems, each CVE required a new application executable because all the code for behavior and interaction was custom to the application. This made deployment and sharing of virtual world difficult because everyone had to have the same executable version. Terra does not suffer from this issue because the virtual world is fully specified in the world model. We only need to write a single viewer that knows how to connect to and interact with the unified world model. This viewer can be reused for any world that has been created. There is no requirement for a custom piece of software for each virtual world. This also means that people could create other viewers that could all interact through this single virtual world representation.

This is similar to how Web browsers have revolutionized the dissemination of documents and deployment of browser-based applications on the Internet. A developer can load a document or application on a web-server and rely upon the fact that existing web browsers will be able to process and display that content. This allows for simultaneous deployment of data and applications to millions of users without requiring them to add anything to their systems.

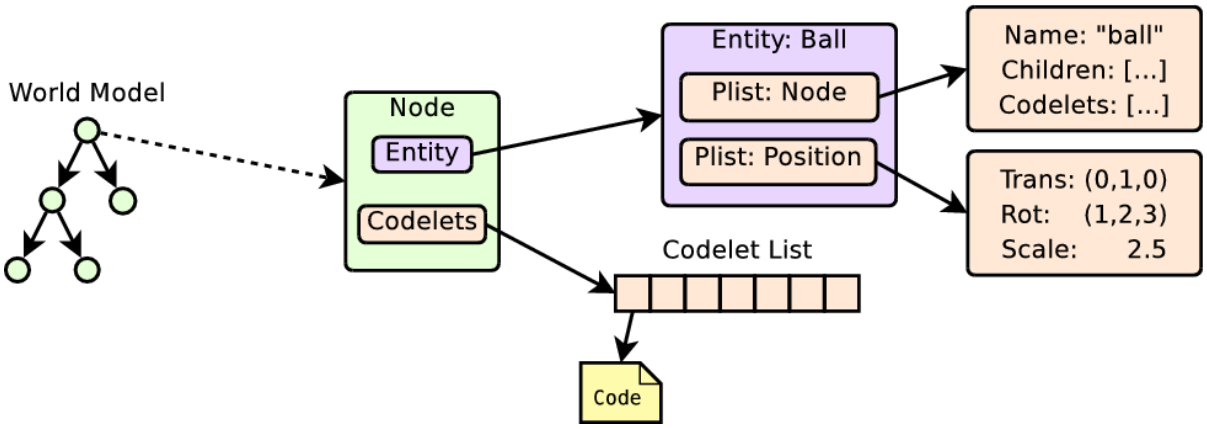

Figure 7.1 Unified world model overview

Terra gives developers of CVEs a great deal of power but behind this layer there is a great deal of complexity. The remainder of this chapter provides an overview of the design of Terra followed by a discussion of how the system has worked in practice.

7.1 Design

The design of Terra can be broken up into three major areas: world model, system managers, and the viewers. We will briefly describe each of these high-level systems and then follow with a detailed discussion of some of the more interesting and involved sub-components of the system.

7.1.1 World Model

The world manager is the core data structure for defining a world in the Terra system. As shown in Figure 7.1 on page 115, the world model is comprised of two major components: nodes and codelets. Nodes represent objects or spatial areas in the virtual world and the data associated with those objects. Codelets are reusable code components that define the behaviors to attach to a given node.

Nodes are realized using a DSO entity that holds a list of node children, a list of attached codelets, and state data for the object represented by the node. Each node can have any

number of other nodes as children but a node can only have one parent. This parent-child relationship defines a directed acyclic graph structure that forms the basis of the relationship of objects within the system and give structure to the virtual world. The class relationships that make up the world model are shown in Figure 7.2 on page 117.

Because nodes are DSO entities, they can use property lists to store any information that needs to be associated with the underlying object. For example if a node represents a car in the virtual world we could associate a property list with the car's color, another one with the car's position, and still another with information about the car's current controls. By using entities as the underlying representation, we allow the user the flexibility to add as much or as little details as is needed for the codelets and other components of the system to represent and interact with that node.

Every Node store a list of associated codelets. Each entry in the codelet list contains two items: a codelet type id and a codelet execution area. The type id specifies the codelet type to look up and instantiate. One way to think of the type is similar to a class type in an object oriented language and then the instantiated codelet a similar to an object. The second piece of information, the execution area, tells the run-time system how to execute the codelet in the distributed system. The codelet could be set to execute on all peers, only on the peer that owns the node, or only on the peers that don't own the node. We will discuss codelets and their execution in more detail in the sections below.

Each codelet can implement any type of unique behavior or interactivity needed by a node. Because codelets are uniquely instantiated for each Node, a single codelet type can be used by multiple nodes in the world model. By reusing and combining codelets, a Node can exhibit very complex behavior without having to create any new codelet types.

7.1.2 System Architecture

The core architecture of Terra is show in Figure 7.3 on page 118. The system structure is based on the ideas of the micro-kernel and mediator design patterns [BMR+96]. The system

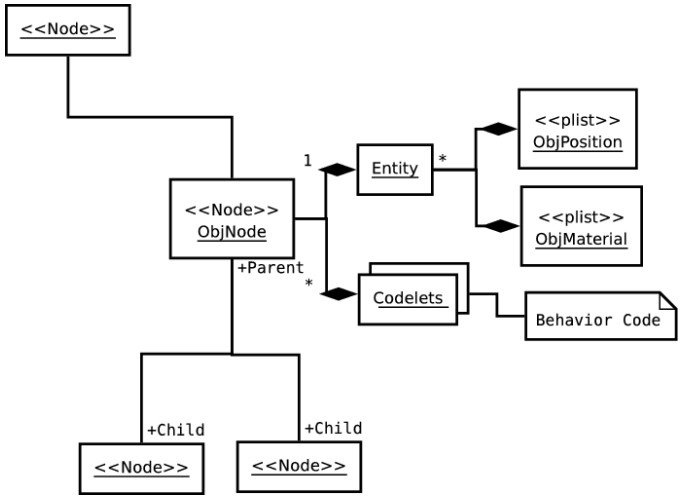

Figure 7.2 Node and codelet relationship in world model

controller is a singleton that acts as the central supervisor for the system. It holds the references to every other component in the system and is responsible for initializing, coordinating, and managing all components and sub-systems. The system controller must control the timing of all the managers in the system to ensure they remain consistent during execution. It keeps track of the user that is logged in and updates the facets that are monitoring the system.

7.1.2.1 SystemController

The system controller bootstraps the system upon start up. First it loads any configuration options that have been given to the system. It holds these options and provides them to later stages of start up and other components in the system. Next the system controller connects to Plexus and DSO to make sure the back-end networking and data sharing are setup correctly. After the networking has been established it can them begin to instantiate and initialize each of the managers in the system in turn. Provided that there are no errors loading the managers, the system controller checks to see if it has been given an initial world configuration. If it has, it triggers the asynchronous loading of the world as the final step in initialization.

Once the system has been initialized it enters a stable frame update state. The SystemController::update() method is called once per frame to update the system. Internally this method coordinates the other components by calling each of them in sequence to update their state.

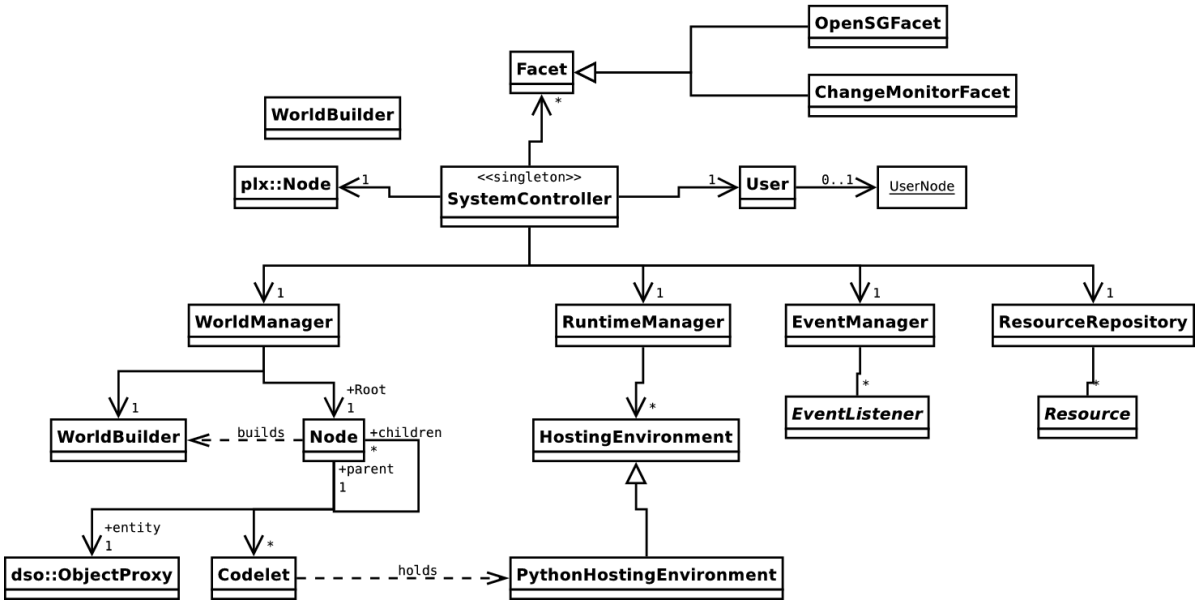

Figure 7.3 Core architecture of Terra

The system controller ensures that each component is updated only at the correct time in the update loop and that each component can rely upon any dependent components being updated as needed. A similar process takes place when the SystemController shuts down. It first makes sure the system is in a consistent state and logs out the user from the virtual world. Then it proceeds to shutdown each of the managers in turn and finally closes all connections related to the SystemController.

The SystemController interface includes a set of central interface for coordinating the logging in and out of a user to the system as well as the creating and loading of virtual worlds. In the case of user management, these methods communicate with the DSO subsystem and manage the state of the User object in the system. Loading and saving is handled by passing through to directly to the WorldManager.

7.1.2.2 WorldManager and WorldBuilder

The WorldManager is in charge of holding and managing the world model. It keeps a reference to the local copy of the world model, provides the central interface for creation and extension of the world, and allows for loading of new worlds. Because the world model is

based on a directed acyclic graph, there is always one "root" node for the entire world. Once we have a reference to this node we can use it to find all other nodes in the system. In practice, when we refer to the id of a world model we are actually referring to the id of this root node. The WorldManager is the place in the Terra system where we hold the base reference to this root node. Any time the rest of the system needs to refer to the world as a whole, it must ask the WorldManager for this node and start from there.

Although the WorldManager is in charge of the world model, it is not the only component of the system that deals with the model. There is a great deal of complexity involved in world construction that is better managed using a separate builder pattern [GHJV95] to keep it encapsulated and separate from the other responsibilities of world management. The component that coordinates and manages the construction the world model is the WorldBuilder. It contains all the code needed to manage the complexity involved in the asynchronous creation, discover, and monitoring of the world model.

For example, at any time during execution a local node in the world model may receive a node update referencing a new nodes in the model. The WorldBuilder monitors these changes. When a change is detected it spawns off asynchronous requests to find these new nodes. When those responses are received, the WorldBuilder integrates the new nodes into the current WorldModel. First it makes sure that the nodes do not conflict with anything locally, then it instantiates a new Node object for the local representation and initializes it with the data from the Node's entity. These changes in turn may contain further changes thus the process can continue recursively until all changes have been followed and the entire world model has been brought into a consistent state locally. This entire process has to proceed asynchronously in response to network responses and must be able to detect and recover from failures. By keeping all of this routines to implement this behavior in one location we attempt to minimize the complexity as much as possible.

When a user needs to create a new Node in the system, they need to coordinate this with the WorldManager and thus in turn with the WorldBuilder so it can run these discovery routines on the new data.

The WorldManager can load a world in one of two ways; it can be loaded by connecting to a remote node or it can be loaded by reading in a local serialized copy of a world that has been previously saved by the WorldManager. In either case, the WorldManager starts the process but relies upon the algorithms and asynchronous behavior of the WorldBuilder to complete the process.

To load a world by connecting to a remote node, we only need to give the WorldManager the id of the root node of the world model. From this id, the WorldManager can spawn a find request to find this node in the virtual world. Once the node is found, then it can use the WorldBuilder to recursively load the rest of the virtual world by triggering the discovery of other nodes rooted at that point much as it would for any new node in the world.

When loading a world from a serialized version, the loading process is slightly more complex. First, the file being loaded must have been previously saved using the WorldManager. It can save a world by serializing every entity used to represent that world. This can entail a large number of interconnected nodes, so the WorldBuilder contains a crawling algorithm that can recurse through a web of entities and build a list of all the entities being referenced. Once this list if built, the system can then save the world by saving all of the entities to a file and storing the id of the root of the virtual world. The entities are saved by serializing them using the generic methods provided by the DSO system for storing entities persistently.

The loading process is very similar to the saving process. The WorldManager first tells the WorldBuilder about the world file and it then loads the data for each serialized entity into an internal cache. This cache maps entity id's to the full data for the entity as it was found in the loaded file. Once the cache loading is complete, the WorldBuilder then looks up the id of the root node for the world and uses this to trigger a world connection much as it does for the standard case. The difference comes in when we fail to find a node on the network. In the standard case, a failure is resolved by leaving a null reference in the world model. But when loading from disk we can handle failures by looking for that node in the cache. If a node is found in the cache then we instantiate a new node using this data and connect it to the existing nodes in the system. This method of loading allows us to use the same code for

both cases and provides flexibility in the case where some parts of the world may be loaded from disk while others are loaded from the network.

7.1.2.3 RuntimeManager and Codelets

Terra uses a component oriented architecture [OCBS99, Szy98, SGM02]for defining the code and behavior in a CVE. The components in the system are called codelets. Each codelet encapsulates a single piece of code that defines a single behavior or algorithms. Codelets can be reused and multiple codelets can be associated with a single node. The aggregate behavior of all the codelets for a node define the behavior of that node in the system. This allows for composability of the codelets to create complex behavior from simple building blocks.

The second part of the component architecture is the RuntimeManager. The RuntimeManager is responsible for loading and managing the codelets along with the hosting environments used for those codelets. It uses hosting environments to load, initialize, and control the various interpreters or run-times that may be needed. For example there is a python hosting environment that loads a python interpreter and manages the loading of python codelets into the interpreter.

The RuntimeManager monitors the world model and recognizes when there are changes to the code in the system. For example when a node adds a new codelets, it is the RuntimeManager that recognizes the addition, finds the codelet resource needed, instantiates a new instance, and connects the codelet to the node.

7.1.2.4 ResourceRepository

The resource repository is in charge of all resource management in the system. It manages the local resource files and communicates with remote nodes to find and retrieve new resources. Terra uses it to find data files that are associated with the world model and tell the systems using it where it is stored so they can load the files.

Terra's resource system is fairly simplistic at it's core. All resources are identified using GUIDs that map to a data file. Terra does not store any information about the type of data

contained in the file or anything about it's format. The files are stored on the local disk in a "resource repository" directory.

Users can add resources to the repository by calling the ResourceRepository with a new resource id and the name of the file for the resource. The system then copies that file to the repository directory, adds the id and reference to the resource map, and starts to serve that resource to any remote repositories that ask for it.

To look up a resource, the user must pass in a resource id. If the resource is already available locally, then the system returns a reference to the file on the local file system. The user can then process this file as it would any other file on the file system.

If the resource was not available locally, then the system will attempt to find it in other remote repositories. The search is done using the same Plexus connection that is used for all other data sharing. When a remote node receives the query it checks if it has the requested resource and if it does, it responds directly to the originating node. Upon receipt of a successful find from a remote node, the local node establishes an HTTP connection to the remote node and begins to download the resource. When the download is complete, the file is moved to the repository and an entry is added to the resource map. The user can then retrieve the resource and load the file as normal.

Because of the way the ResourceManager works, there is an absolute requirement of unique identifiers. If the same id is used on multiple nodes for different resources, the behavior of the system in undefined. One interesting implication of this requirement is that when a user wants to upload a new version of an existing resource, they actually have to create an entirely new resource with a new id and change all references to the old resource to point at the new resource. This is needed in order to provide a guarantee that all nodes have the exact resource files they think they should.

7.1.2.5 User

The User object is the representation of the local user in the virtual world. It controls the entity representing the local user in the world model and thus completely controls the virtual

embodiment of the user within the virtual world. As the user moves through the scene and interacts with objects in the environment we update the User object so both the remote viewer and the local viewer maintain a consistent description of the users state. One piece of state held in the User object is their current location in the virtual world. As the user manipulates the local viewer application this causes the location information in the User object to change. This change is then used to immediately update the local viewer and is used to update the state information for the local user in the world model.

Normally a remote user is represented using an avatar. There is one avatar per remote user and it is mapped onto the information in the entity controlled by the remote User object. The entity holds not only the location of the user, but also additional useful information such as their view direction, where they are pointing, and anything else needed to update their avatar state to be viewed remotely.

It is important to communicate this state information because an accurate remote representation is key to a productive and useful collaborative experience. We do not have a "real" view of the remote user, but by modeling their avatar as closely as possible to the real-world behavior we are able to translate a great deal of non-verbal communication keys that can be very important to human interaction. Knowing something as simple as knowing the direction that the user is looking or what they are pointing at in the scene is very important for collaboration. For example in a collaborative design review scenario multiple users come together in a shared space to review and discuss prospective designs that are represented by objects in the virtual scene. As the users talk about the objects they frequently say things such as "what is the problem here" or "look at this". Without knowing where they are looking or pointing, it is nearly impossible to have a constructive design review because nearly all of the interpersonal communication would need to be spent describing what aspects of the scene they are discussing.

When the local user is represented remotely as an avatar, the system uses a standard model file to represent the user. The model file to use must be a resource in the system. The exact resource used is specified by setting the user model property of the entity associated with the

User. This tells the remote node what avatar to use for showing the user and allows user's to customize their appearance. This becomes more important as more user enter the system and the avatar is used to recognize the remote user. For example if there are eight people in a collaboration and they each use the default user avatar it is very difficult to visually "find" the person that you want to talk with in the environment. Alternatively if each user specifies a different model and they use that model ever time they collaborate, then people can recognize each other through their avatars. Once again, this is a simple capability but it has a dramatic impact on the level of engagement and collaboration in the system.

The User object manages the authentication of the local user to the system. These capabilities are provided by using the SecurityManager from DSO. To login to the system, the user must specify their name and their key information. Terra then uses this to connect to DSO and begin exchanging data. We anticipate that this portion of the system will be extended a great deal as part of future work to provide for better user management and more extensive support for authentication and security.

7.1.2.6 Facets

When using a world model and presenting it to the user, there is often a need to create new representations derived from the world model. For example, we may need a custom representation to display the world in a graphical 3D environment, to interface with a physics simulation, or to present the model for editing in a desktop GUI. In all these cases we have a new data structure that reflects a different view of the world model using a perspective that is most appropriate for the final use. We describe these derived perspectives as *Facets*. It is through facets that the user and other parts of the system are able to perceive and interact with the world model.

All facets share a few common design characteristics. Because they are providing a different way to represent the world model, they must monitor the world model and reflect all changes that occur. The system design helps support this by calling an update method on all facets once per frame. The implementation of this update method can then use the change

monitoring support from DSO and Terra to continually make refinements to the facet's data structures based upon the changes observed. For example if a node representing a ball in the world has it's color property changes, then any facets that are presenting data visually must see this change and modify the color being rendered.

Facets are often used as bridges to different subsystems that need to connect to the world model. In this way they can be looked at as a realization of a facade design pattern [GHJV95]. They present the world in a way that allows other parts of the system to see what they need and to interact with the world without having to know all the details of the full world model. For example, a physics simulation may want to see all objects in the world as independent entities with Newtonian properties. This is not how the world model represents the objects but by using a facet we can provide a perspective that presents the world in exactly this way. By using facets we are simultaneously make it easier for extensions to be added to the system and providing explicit points of coordination in the system. We will discuss facets in further details later in this chapter.

Current Facets In the current system we have designed and implemented several facets. The first facet is used for a desktop UI to edit the world model. This facet allows the user to explore the entire tree structure of the world model. They can see how the nodes relate to their children. When the user selects a node, they can see all the properties of that node and all the data contained in the entity and associated property lists for the node. In addition to seeing this data, the user can also edit the world model. They can add, remove, and modify nodes, property lists, and codelets.

Although presenting the world model in this way can be a bit overwhelming, it gives the user a great deal of power. Through this interface they can browse existing worlds, create new worlds, and debug developing worlds. This facet provides so much flexibility that it currently serves as the primary development platform for all of the worlds we have created.

The second facet in the current system is used for visual representation and is called the OpenSGFacet. It is named this because it provides the link connecting the world model to the OpenSG scenegraph [Rei02] that we use to present the world in 3D environments. This facet

is responsible for turning the world model and all it's state into a scenegraph representation that we can render for the user. This entails everything from loading 3D models to rendering the user's avatar in the scene.

7.1.3 Viewer

The Terra libraries provide a great deal of capabilities, but we still need an interface to bring these capabilities together and present them to the user in a useful way. This is where the viewer application comes into the picture. We could write any number of viewers on top of Terra and each could provide different views into the system, but we have only created two for now. Both viewers are named lglass, but each is a variant that is targeted for a different presentation medium. One is meant for the standard desktop interface and the other is meant for immersive VR systems.

The viewers are based upon a common core and shared much of the same code-base. The common core is based on Plexus, DSO, Terra, and the OpenSGFacet. It implements a common metaphor of moving a virtual user through the virtual world represented by the current world model. As the user moves through the world their current location information is passed into the world model using the User object that represents them in the virtual world. The code used for presenting the visuals of this 3D environment are common to both viewers and is nearly entirely captured in the facets used by the viewer. The place where the viewers differ is in how they present the user interface and in how they allow the user to interact with the virtual world.

The desktop viewer presents a classic desktop-style interface and interaction paradigm. It uses the PyQt windowing system [1] to provide a view into the world model, a simple interaction model, and some basic scene editing capabilities. The interaction model consists of using the mouse to select, drag, and control objects in the scene. This method used will be discussed in more detail later in this chapter.

The immersive viewer provides an interface using the VR Juggler library and can be used

[1] http://www.riverbankcomputing.co.uk/pyqt/

across clusters of machines for large VR systems. Like the desktop viewer, it presents a view of the world model and allows the user interact with the scene. Where it differs is in how the interaction is handled. In the VR system all interaction is done using a virtual wand and natural grabbing and selection methods.

7.1.3.1 EventManager

The EventManager is the central point in the system for all interaction or other events that influence the evaluation of the code in the world model. The design of the EventManager is based on an observer pattern.

Any object in the system can register to monitor events in the system. All events are identified by two pieces of information: a node id and a signal name. The node id is a GUID that corresponds to the id of a node in the world model. The signal name is a user defined string. Any name may be used and new system components may introduce events that are specific to their functionality. This provides a point of extension that allows for any number of signal types to be used without ever having to modify or extend the EventManager itself.

Users register to observe an event by calling the EventManager with the id of the node to monitor, the name of the signal to monitor, and a callback. When a system component needs to emit a signal they call the EventManager with the id of the affected node, the name of the signal being emitted, and a list of arguments to pass to the handlers. The EventManager then looks up all observing callbacks and calls them with the arguments adapted as needed for the callables.

The EventManager also controls the timers in the system. A user can register a periodic callback by adding a timer to the EventManager. The EventManager will then call the timer and callback as needed. Arguments and connections are handle similarly to the method use for the standard events.

The EventManager provides the benefit of flexibility and extensibility to the system. By providing a standard way to communicate between components, we decrease the complexity of each component. They do not have to re-invent a new observer system or find a new way to

pass data to interested users. Instead they can use the common standard to meet their needs and extend it as needed for all their communication. This has the added benefit that it allows all components in the system to stay relatively decoupled from each other because they do not have to call each other directly. Instead they can use the EventManager to communicate indirectly by using known signal names to call any number of unknown observers. For example a viewer application can monitor a user's interaction and send a signal that they selected an object in the scene. A codelet can pick up this signal and process it in anyway the see fit. This entire process takes place without the viewer having to know anything about the interface of the codelet and without any direct communication. Taken a step further, this process can even be used to allow two dynamically loaded codelets, from different users, that have been developed completely independently, to communicate with each other without ever seeing each others code.

7.1.3.2 Interaction

We needed a way to capture the various interaction paradigms in a reusable way that could be applied to many different codelets and could be mapped from various viewers. At first we investigated allowing direct access to the various user interfaces provided by the viewers. For example the code could look at mouse movements on the desktop and wand movements in a virtual environment. It quickly became apparent that this would not scale well. This paradigm results in a great deal of code in the codelets that is just used to decipher what input devices are available and then to interpret them. We knew that this would not work so we found another way.

What we ended up with was an event based systems that abstracted all virtual interaction into a well-defined set of interaction events. This systems makes use of the capabilities provided by the EventManager to route and process events for each node in the world model. In the case of interaction, the viewer application is responsible for translating user interaction into interaction events for each node.

Currently the system supports the following interaction events:

point, point_start, point_stop: These events are fired when the user points to an object in the scene. This may be with a virtual wand or by hovering their mouse in desktop mode.

touch, touch_start, touch_stop: These events are fired when the user brings their interaction device into contact with the object in the virtual environment. For example in an immersive environment this event is sent when the user's wand intersects with the virtual object.

move, grab_start, grab_stop: These events are fired when the user is attempting to grab and possibly move an object in the scene.

Codelets respond to user interaction by registering their interest to these events on a given node with the EventManager. For example if a codelet wanted to change the color of an object when the user touched an object in the scene, it would register the "touch_start" and "touch_stop" event for the node of interest with the EventManager. Then when the user touched the node, the EventManager would call the event management code in the codelet and the codelet could then change the color property of the object. This code would work in both the desktop environment and the immersive environment.

Although we currently restrict the interaction to these three groups of interaction types, we have found that they provide for the vast majority of needed interactions. This part of the system could be extended in the future to support other interaction needs as they become apparent.

7.2 Discussion

During the development of Terra we encountered many challenges. We tried many alternatives before arriving at the system presented in the previous sections. Fortunately we were able to find suitable methods to address each challenge. Some of the more interesting challenges are discussed below.

World Manager and Builder The sharing of responsibilities between he world manager and world builder is currently a bit problematic. In the initial design, the idea was that the WorldManager would own the world and that the WorldBuilder would be responsible for helping to load and build the world. When it came to implementation, this simple split became much more complex. In the current realization there are several work-arounds contained within the WorldBuilder that allows it to build the world and callback into itself when a new node is added. For example, asynchronous detection and building of new world nodes and/or trees proved to be very complex to implement. The WorldBuilder has to know a significant amount of information and effectively takes over the management of the world model during several phases of construction. This relationship should receive more investigation in the future to see if there is a more elegant way to handle the sharing of responsibilities.

Codelets We originally planned to base our codelet system off .Net components. The common language run-time seemed to provide a very nice component architecture to support this capability. This relied upon finding a way to link the CLR components to the Terra runtime and as it ended up this proved to be very difficult and would have required extensive additional work. Instead of pursuing this work we decided to use Python which already has a very straight forward interface to C/C++ code. As it turns out, this choice was very fortunate because it allowed us to take advantage of the dynamic capabilities of Python from within the Terra system. In the future other people could revisit this choice and add support for additional languages and run-times.

One area where we spent a good deal of thought was in whether to allow codelets to interact with each other directly. In our original design we planned to keep all codelets completely independent and only allow cross-codelet communication by way of data updates to the entities. This choice was made because we thought it would ensure that codelets were completely encapsulated and reusable. When we began to implement various virtual worlds though we realized that in some cases inter-codelet communication was very helpful and that the overhead of doing this through property lists could be to great. We were still very hesitant to allow direct communication though. Instead we began to experiment with using the Event-

Manager to send events between various codelets. This provided a way to allow codelets to call each other without directly exposing references or APIs for specific codelets. So far this system has worked well and has provided for all the inter-codelet communication necessary. In the future this decision could be revisited to see if there is any other way to allow codelets to collaborate more directly.

Property Lists The Entity and PropertyList basis for Nodes provides a great deal of flexibility, but this has to be tempered with the fact that Codelets need to know the data structure to communicate. One slightly unexpected side-effect of this requirement is that the number of PropertyList types remained fairly low for most CVEs. Through the course of developing virtual worlds, the development of codelets that could work on multiple nodes not only drove the requirements of the property list types, but it also drove towards a standardization of the types. This was needed because many codelets needed to be written and it simply made sense that they would want to use similar data structures. What we found worked very well was to think of objects from the perspective of physical properties and then break these properties down into groups of similar properties. Each group then because a single property list. For example, position, size, and scale were grouped into one property list type that was used for all objects that had a physical realization in the environment. If an object had editable material properties, then another property list may be added that would describe the characteristics such as color. As people continue to use this system, we anticipate that an entire taxonomy of such property list types will be developed to provide standard ways for codelets to interact and interpret the characteristics of objects.

Facets Implementing the idea of facets proved to be very problematic. The core issue is that a facet is really a new data structure that reflects an existing data structure. Because of this, the code must monitor changes in the source data structure (the world model) and then make iterative changes to the local data structure. This may sound simple, but in practice it can be very difficult to keep the data structures in sync. What we had to do for the facets we have implemented is keep back references into the facet data structure so for example at any

time we can find the portion of the data structure that corresponds to a given node.

Even more difficult then this though is that the data structure may reflect characteristics from the properties in the node entity's property lists. For example the OpenSGFacet has a set of scene graph nodes for each node in the world model. If the world model has data about transformations, then we have a scene graph transformation nodes, if there are material properties, then we have a material node for the corresponding geometry, and so on. This can lead to an ever increasing set of scene graph nodes to represent a single world model node. As of yet we have not found a way to simplify this complexity.

Facets also prove difficult when we take into account that they may need to interact. For example a physics facet may need to know the geometry of a scene in order to correctly represent the objects for collision detection. This requires the physics facet to examine and monitor not only the world model, but also the OpenSGFacet that contains the real geometry. The current design does not address this flow of data and dependency between facets. Future work is needed to find a way to manage these inter dependencies.

Resource Management The distribution of resources to a node is currently a bit simplistic, but it seems to work well. We originally thought we could take advantage of parallel streaming of resources from multiple connected users to increase the speed of resource downloads. We had hoped to build off of a system such as BitTorrent [Coh03]. Unfortunately we found that not only did this make the system more complex, but upon further analysis it did not appear that it would have a significant impact on performance. Most objects in the scene are very small and the overhead of managing parallel downloads, distributed trackers, and the increase in network traffic did not seem to warrant the extra complexity. Instead we decided to use a simple HTTP download server local to each connected user. This keeps the system very simple and still allows for parallel download of data by pulling one resource for Node A and another resource for Node B. It may be worth re-examining this decision in the future if resource download times become an issues.

8 Conclusions and Future Work

In the previous chapters we have described a system architecture that can support a new type of CVE creation and deployment. In sections 1.3 through 1.5 we laid out the research problem to address along with the challenges and research issues this work would need to answer. In this section we will describe how each of those issues have been addressed. We follow this with a description of the outcomes of this research and a summary of future work that we feel could further extend the state of the art in CVEs.

8.1 Challenges Addressed

As discussed in Section 1.3, there are many challenges faced by CVE software systems. We set out to address several software related issues in this research. In this section we will discuss how each of these issues relates to the our research.

- Lack of common model of the virtual world

  This is the primary research challenge addressed by this work. Continuum and more specifically Terra provides a single unified world model. It provides a common method for representing all object data and application code within and between CVEs. This model allows for reusability of code and resources across many different application areas and viewers.

- Virtual worlds are not user extensible

  The unified world model provided in Continuum empowers all users to contribute equally to the CVEs created. It allows for full *evolvable extension*. Any user of the system can extend the system with new object data and/or code in anyway they like. There is

no central authority that controls the allowable content or that controls the simulation that can be executed. Similar to the world wide web, users are fully in control and can provide any content that they wish.

- Deployment is difficult

  Because of the way the world model is used, Continuum makes deployment fully automatic. Deployment occurs as a side-effect to standard world model loading. When a user connects to a virtual world using a viewer, the viewer finds the root node of the relevant world model and then begins to traverse the world model from this point. Initially, there is nothing beyond the base viewer and root node. As the viewer discovers new nodes the content and code for these nodes are dynamically downloaded and added to the local world representation. Everything that makes up the world is downloaded on demand. When an update or change is made to a virtual world this is automatically detected and the new state is picked up by all viewers. The entire process of deployment is automatic and transparent to the user.

- Application development is burdensome

  Continuum helps to alleviate some of the development burden by providing higher-level abstractions that hide many of the low-level details that make developing CVEs so complex. Developers do not need to worry about networking, sharing data, keeping state synchronized, deploying applications, or distributing resources. That is all handled by the Plexus, DSO, and Terra layers. Instead developers are free to focus on creating engaging content and interesting behaviors using the world model and Codelets. Although this system is still not without complexity, it has proven to greatly simplify the number of issues that developers have to contend with while creating CVEs. We believe that development could be further simplified in the future through the creation of better tools and development environments for interacting with and creation content inside the world model.

- Limited number of users

Continuum addresses the limits on the number of users by creating a system that allows for user extension and a shared world model. The unified world model allows for CVEs that are composed of contributions from large number of users. However it does not solve the more general problem of network and system scalability. We feel that although this is an important topic, it is outside the scope of this research. We are hopeful that others will follow this research and find ways to use the ideas we have proposed in combination with techniques such as multi-level area of interest models and dynamic network groups to create systems that are more scalable.

8.2 Research Issues Handled

In addition to the general CVE challenges proposed, there were specific research issues brought up in Section 1.5. We described how each of these issues interrelate with each other and the challenges to CVEs. We will reexamine each of them here in the context of how the design of Continuum addresses each issue.

- Unified World Model

  As described in Chapter 7, the Terra sub-system provides a complete world model. It holds object state information in DSO entities and behavior in Codelets. This system provides the backbone for all CVEs built with Continuum.

- Data Distribution

  Terra uses DSO for all data distribution. The DSO system allows developers to create property list that hold the full state for a given aspect of an object. When an instance of this property list is created and attached to a Node's entity, DSO ensures that all object state is distributed to all other users in the system. This includes keeping the data in sync, controlling access rights, and distributing property list type information.

- Code Management

  The RuntimeManager in Terra works in concert with the various HostingEnvironment instances to control all issues related to code management (see Section 7.1.2). Code

components are represented by Codelets that are loaded by the RuntimeManager. Each Codelet is hosted by a HostingEnvironment that provides the environment and bindings needed for whatever language or run-time is required by the given Codelet. The system provides basic support for continuous execution using hooks that allow Codelets to save and restore their state when a new version is found or the system must be reloaded.

One area of code management that requires further work is that of code security. The current hosting environments do not do much in the way of restricting execution or looking for malicious code. This needs to be addressed before Continuum could be widely deployed in unsecured environments.

- Resource Management

  The ResourceManager in Terra is responsible for all resource management in the system. As we have stated previously, we chose to implement a very simplistic model of resource management that is based on unique ids for all resources and a simple find and download pattern for acquiring a needed resource. This works well in practices and addresses most of the requirements for resource management. Version management is provided by use of unique ids. Any resource added to the system is a specific version with a specific id. If a new version is needed, then the objects that reference the resource reference the new id and that id takes precedence.

  One weakness in the current system is that there is no form of authentication for the resources. This is an area for further extension and we believe that this could be easily addressed by using some type of signature system where by a resource id would be a combination of a unique id and a hash to provide a level of confidence in authenticity.

- Space Sharing Rules

  From the outset of this research we knew that one unique issue would be that of space sharing rules. Because our system provides for distributed user extensibility, this introduces issues of who has rights to spaces in the CVE and who can extend the CVE in a

given area. The current system handles these issues through the use of access control rights to node data. Because each node can only be edited and extended by a given set of users, this restricts who has rights to given spaces in the system.

Because of the complexity of the core research into the world model, we have not put as much focus on this area as we would have liked. That said, we have provided the basis for some experimentation and we look forward to seeing how this part of the system could be extended and refined in the future.

- Development Tools

  We have said from the outset that we would like to provide better tools to developers of CVEs. Currently, the primary development tool in Continuum is the desktop viewer. This viewer allows developers to view and modify the world model at run-time. Developers can add new resources, edit node state information, introduce new codelets, and refine existing codelets. The effects of these changes are able to be seen immediately in the system. Through this work, we believe that we have made progress on this front but there is still much work to be done.

8.3 Outcomes

The outcome we are most pleased with is that the system works and it has been deployed in research and production settings. The design not only looks good on paper, but it can be implemented and fielded for real work. This required more effort on our part but it was worth it because it helped us refine our ideas and prove that they work for real CVEs.

8.3.1 Impact

As we described above, we have addressed all of the research issues and challenges that we originally tackled. While we are proud of these accomplishments there are a few that are worth further discussion because they provide the most impact in this research area.

We have shown that this method of structuring a world has promise. It works and it can be used. Before this effort we did not know if this idea had any merit. We have shown that although centralized control of CVEs is the standard, it is not the only way to structure these environments. This simple act of testing a new method of structuring a CVE may have the most long lasting impact. It is our hope that this will serve as a catalyst for more research into this area so future researchers can take this ideas and push them even further.

We have overcome many of the limits of centrally controlled server-based CVE systems. Foremost, the system allows for collaborative creation, editing, and extension of CVEs; what we call evolvable extension. Users can add to the virtual world as they see fit and the world model will take care of merging all these changes into one coherent shared environment. This frees users from the shackles of having a single central authority with a set of servers controlling the entire system.

Deployment of these worlds is straight forward and works as part of the standard system. Much like the world wide web, once you have a viewer application you can simply connect to the world you want and everything needed for presenting the world will be downloaded from that initial connection. This eliminates a significant problem of previous CVE efforts.

8.3.2 Limitations

We would be remiss to suggest that everything is fully solved. There are limits to our current design and there are still many areas that need further research.

Development is still rather difficult. The use of scripting languages has made development more approachable, but there need to be better tools for developing virtual worlds. We would like to see capabilities added such as an integrated code editor, a codelet debugger, and support for developer tracing of state changes and networking. It is our experience that no matter how much developer support a system has, there can always be more added to make the system better. We leave it to future CVE developers to find these limits and extend the tools to suit their needs.

The inherent limits of networking are still an ever present hindrance to scalability. In

a peer-to-peer system like Terra this is especially apparent because the simulation may be widely distributed. The latency between nodes can cause users of simulations to lose their suspension of disbelief. In centralized systems the simulation components are usually on a local high-speed network that can help to reduce the network effects. By distributing the simulation across all users of the system we have made the virtual world more extensible but a side-effect of this is that network performance has more of an impact. This is a trade off that we have not been able to overcome as of yet. We believe there are solutions to be found in the way the Plexus network and thus the DSO communication is managed and routed.

A significant limit of the current system is the lack of a full security infrastructure. We had hoped to design and test a full set of security policies for areas of the system such as user authentication, object state communication, resource management, and code execution. In the end we have plans on the drawing board but we did not have the resources to test and refine the ideas through implementation. As far as we know there is no part of the system that could not be secured with more time, and we think this would be a great project for a student to pursue in the future.

Rather then looking at these limits as shortcomings in our research, we look at them as opportunities for other researchers in the future. In the next section we will describe future work that we think could be done to solve these issues and extend the system further.

8.4 Future research

One area that can always use further research is scalability. We have created a world model that can be used by multiple users, but we have not addressed how to deal with the issues of scalability as large numbers of users interact with this world model. There has been much previous work in this area and I am sure it will continue to be an area of active research.

The networking algorithms in Plexus could be refined further. The current system is very simplistic in it's approach to balancing the network and reducing the number of communication links. We experimented with a few more advanced routing algorithms based on genetic algorithms and other related techniques but we did not pursue these areas nearly far enough.

As we pointed out in the previous section, we need to added a consistent security infrastructure to the entire system. We need a way to authenticate users, the data being exchanged, and the resources being used. We believe that this system could be built upon existing best practices.

We have only begun to scratch the surface of what is possible with Codelets. There is a significant amount of work that is needed to investigate issues such as inter-Codelet communication, user interaction, persistence, and long-term viability of applications. Once more applications are written using a model such as this one, we believe that issues related to Codelets and application development will be a fertile area for future projects.

As with all research projects, we have not reached the ultimate solution for any areas. As such, every research area addressed in this project could still use further research. Our simple hope is that we have provided a starting point that will plant the seed for future researchers to follow this work and extend it with their own ideas.

8.5 Final thoughts

This research work has been simultaneously more difficult and more rewarding then originally planned. Because of the complexity of the problems addressed it has taken a significant amount of effort to create the designs needed for the system and test them out in the real world to make sure they work. This second step of realizing the design ideas with real working code has taken the greatest amount of time, but the effort has paid off in that it required us to think through the issues more clearly. In some cases early ideas did not work out and we had to revisit the designs, in other cases we were able to come up with new design ideas based upon the feedback from implementing the system. We are very happy with the outcomes of this work. We have learned a great deal and we believe we have extended the knowledge in this area and provided a springboard for future research in this area.

Overall the design of Continuum has worked very well. It has allowed us to create complex worlds out of simple components and data descriptions. The system has proven itself in the real-world but there is still much work yet to be done. We do not consider this system

to be complete, we consider it to be just complete enough. We are excited to be a part of this field and we are very interested to see where the next line of researchers can take this effort. We hope that in the near future there will be vast CVEs as available and extensible for users as modern day websites our to their users.

BIBLIOGRAPHY

[Ada01] D. Adams, *Programming Jabber*. O'Reilly, 2001.

[AG98] K. Arnold and J. Gosling, *The Java Programming Language*, 2nd ed. New York, NY: Addison-Wesley Publishing Company, 1998.

[AL01] P. Albitz and C. Liu, *DNS and BIND*, 4th ed. O'Reilly and Associates, 2001.

[bam02] (2002, Nov) Bamboo website. [Online]. Available: www.watsen.net/Bamboo

[BBFG94] S. Benford, J. Bowers, L. Fahlen, and C. Greenhalgh, "Managing mutual awareness in collaborative virtual environments," in *Virtual Reality Software and Technology (Proceedings of VRST'94, August 23-26, 1994, Singapore)*, Singapore, 1994, pp. 223–236. [Online]. Available: citeseer.nj.nec.com/benford94managing.html

[Bec01] K. Beck, *Extreme Programming Explained*, ser. The XP Series. Addison-Wesley, 2001.

[BF93] S. Benford and L. E. Fahlen, "A spatial model of interaction in large virtual environments," in *Third European Conference on Computer Supported Cooperative Workd*, Milano, Italy, Sept 1993, p. 107. [Online]. Available: citeseer.nj.nec.com/benford93spatial.html

[BG97] S. Benford and C. Greenhalgh, "Introducing third party objects into the spatial model of interaction," in *ECSCW*, 1997, pp. 189–. [Online]. Available: citeseer.nj.nec.com/benford97introducing.html

[Bie00] A. Bierbaum, "VR Juggler: A virtual platform for virtual reality application development," Master's thesis, Iowa State University, Ames, Iowa, 2000.

[BJH+01] A. Bierbaum, C. Just, P. Hartling, K. Meinert, A. Baker, and C. Cruz-Neira, "VR Juggler: A virtual platform for virtual reality application development," in *Proceedings of IEEE Virtual Reality*, Yokohama, Japan, March 2001, pp. 89–96.

[BMR+96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture Volume 1: A System of Patterns*. Wiley, Aug. 1996.

[Bou02] D. Bourg, *Physics for Game Developers*. O'Reilly and Associates, 2002.

[Box97] D. Box, *Essential COM*. Addison-Wesley, 1997.

[BZWM97] D. P. Brutzman, M. Zyda, K. Watsen, and M. R. Macedonia, "Virtual reality transfer protocol (vrtp) design rationale," in *Proceedings of Workshops on Enabling Technology: Infrastructure for Collaborative Enterprises*. Cambridge, Massachusetts, United States: IEEE Computer Society, 1997, pp. 179–186.

[CDK01] G. Coulouris, J. Dollimore, and T. Kindberg, *Distributed Systems: Concepts and Design*, 3rd ed. Addison-Wesley, 2001.

[Coh03] B. Cohen, "Incentives build robustness in bittorrent," *Workshop on Economics of Peer-to-Peer Systems*, 2003.

[CP01] J. Cook and S. Pettifer, "Placeworld: An integration of shared virtual worlds," in *SIGGRAPH 2001 Sketches and Applications Programme*. ACM Press, Aug 2001, p. 234.

[DeL00] M. DeLoura, *Game Programming Gems*. Charles River Media, Inc., 2000.

[DeL01] ——, *Game Programming Gems 2*. Charles River Media, Inc., 2001.

[Div03] "Dive homepage," Jan 2003. [Online]. Available: http://www.sics.se/dive/

[Ebe01] D. H. Eberly, *3D Game Engine Design: A practical approach to real-time computer graphics*. Morgan Kaufmann, 2001.

[ECM02] ECMA International, *Standard ECMA-334, C# Language Specification*, 2nd ed. ECMA International, December 2002.

[Eve] "Everquest homepage." [Online]. Available: http://everquest.station.sony.com/

[Fit99] C. Fitch, "Strategies for scaling interactive entertainment," *5th International Conference on Virtual Systems and Multimedia*, 1999.

[FJL+97] S. Floyd, V. Jacobson, C. Liu, S. McCanne, and L. Zhang, "A reliable multicast framework for light-weight sessions and application level framing," *IEEE/ACM Transactions on Networking*, vol. 5, no. 6, pp. 784–803, December 1997. [Online]. Available: http://www.icir.org/floyd/srm-paper.html

[Fol90] J. Foley, *Computer graphics: principles and practice*. Addison-Wesley, 1990.

[Fow97] M. Fowler, "Dealing with properties," 1997, from martinfowler.com. [Online]. Available: http://martinfowler.com/apsupp/properties.pdf

[Fre03] "Freenet homepage," Jan 2003. [Online]. Available: www.freenetproject.org

[FS98] E. Frecon and M. Stenius, "Dive: A scalable network architecture for distributed virtual environments," *Distributed Entineering Journal (special issue on Distributed Virtual Environments)*, vol. 5, no. 3, pp. 91–100, Sept 1998. [Online]. Available: www.sics.se/~emmanuel/publication/dsej/dsej.html

[FY98] B. Foote and J. W. Yoder, "Metadata and active object-models," *Fifth Conference on Pattern Languages of Programs*, Aug 1998. [Online]. Available: http://www.laputan.org/metadata/metadata.html

[GB95a] C. M. Greenhalgh and S. Benford, "Massive: A virtual reality system for teleconferencing," in *ACM Transactions on Computer Human Interfaces*. ACM Press, Sept 1995, pp. 239–261.

[GB95b] C. Greenhalgh and S. Benford, "MASSIVE: A distributed virtual reality system incorporating spatial trading," in *International Conference on Distributed Computing Systems*, 1995, pp. 27–34. [Online]. Available: citeseer.nj.nec.com/greenhalgh95massive.html

[Gee00] C. Geenhalgh, "Massive-3 user guide," Dec 2000. [Online]. Available: http://www.crg.cs.nott.ac.uk/research/systems/MASSIVE-3/docs/massive3-user.html

[Gel85] D. Gelernter, "Generative communication in linda," *ACM Transactions of Programming Languages and Systems*, vol. 7, no. 1, jan 1985.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, ser. Addison-Wesley Professional Computing Series. New York, NY: Addison-Wesley Publishing Company, 1995.

[Gnu03] "Gnutella.com," Jan 2003. [Online]. Available: www.gnutella.com

[Gre99a] C. Greenhalgh, "Massive-3/hivek application programming: Basics," Jun 1999. [Online]. Available: http://www.crg.cs.nott.ac.uk/research/systems/MASSIVE-3/docs/massive3-app-prog-intro.pdf

[Gre99b] ——. (1999, Jun) Massive-3/hivel introduction. [Online]. Available: http://www.crg.cs.nott.ac.uk/research/systems/MASSIVE-3/docs/massive3-intro-summary.pdf

[Ham96] G. Hamilton, "Javabeans api specification," 1996. [Online]. Available: http://java.sun.com/products/javabeans/docs/spec.html

[Har01] P. L. Hartling, "Octopus: A study in collaborative virtual environment implementation," Master's thesis, Iowa State University, Ames, Iowa, 2001.

[HV99] M. Henning and S. Vinoski, *Advanced CORBA Programming with C++*. Addison-Wesley, 1999.

[HW96] J. Hartman and J. Wernecke, *The VRML 2.0 Handbook*. Addion-Wesley, 1996.

[Kob87] N. Koblitz, "Elliptic curve cryptosystems," *Mathematics of Computation*, vol. 48, pp. 203–209, 1987.

[Lab] L. Labs. Second life. [Online]. Available: http://www.secondlife.com

[Lin] "Lineage homepage." [Online]. Available: http://www.lineage.com/

[Liv03] "Living worlds working group," Apr 2003. [Online]. Available: www.vrml.org/WorkingGroups/living-worlds

[Mas] "Massive-3 homepage." [Online]. Available: http://www.crg.cs.nott.ac.uk/research/systems/MASSIVE-3/

[Mil85] V. Miller, "Use of elliptic curves in cryptography," *CRYPTO*, no. 85, 1985.

[Mol99] T. Moller, *Real-time rendering*. A K Peters, Ltd, 1999.

[Moo65] G. Moore, "Cramming more components onto integrated circuits," *Electronics Magazine*, vol. 38, no. 8, Apr. 1965. [Online]. Available: ftp://download.intel.com/museum/Moores_Law/Articles-Press_Releases/Gordon_Moore_1965_Article.pdf

[MS01] E. Meijer and C. Szyperski, "What's in a name? .NET as a component framework (invited paper)," in *Proceedings of First OOPSLA Workshop on Language Mechanisms for Programming Software Components*, Tampa Bay, Florida, United States, October 2001, pp. 22–28.

[Obj02] Object Management Group, *The Common Object Request Broker Architecture: Core Specification*, 3rd ed. Object Management Group, December 2002. [Online]. Available: http://www.omg.org/cgi-bin/doc?formal/02-12-02.pdf

[OCBS99] M. Oliveira, J. Crowcroft, D. Brutzman, and M. Slater, "Components for distributed virtual environments," in *Proceedings of the ACM Symposium on Virtual*

Reality Software and Technology. London, United Kingdom: ACM Press, 1999, pp. 176–177.

[OCS00] M. Oliveira, J. Crowcroft, and M. Slater, "Component framework infrastructure for virtual environments," in *Proceedings of the Third International Conference on Collaborative Virtual Environments*. San Francisco, California, United States: ACM Press, 2000, pp. 139–146.

[ope] Openid homepage. [Online]. Available: http://openid.net

[Ora01] A. Oram, *Peer-to-Peer: Harnessing the Power of Disruptive Technologies*. O'Reilly, 2001.

[PCMW00] S. Petttifer, J. Cook, J. Marsh, and A. West, "Deva3: Architecture for a large scale virtual reality system," in *Proceedings of ACM Symposium in Virtual Reality Software and Technology*. ACM Press, Oct 2000, pp. 33–39.

[Pet99] S. Pettifer, "An operating environment for large scale virtual reality," Ph.D. dissertation, University of Manchester, April 1999.

[Pri99] J. Pritchard, *COM and CORBA Side by Side: Architectures, Strategies, and Implementations*. Addison-Wesley, 1999.

[Rei02] D. Reiners, "Opensg: A scene graph system for flexible and efficient realtime rendering for virtual and augmented reality applications," Ph.D. dissertation, Technical University Darmstadt, Jun. 2002.

[Rog97] D. Rogerson, *Inside COM*, ser. Programming Series. Microsoft Press, February 1997.

[RS97] D. Roberts and P. Sharkey, "Maximising concurrency and sclability in a consistent, causal, distributed virtual reality system, whilst minimising the effect of network delays," in *Proceedings of the 6th International Workshop on Enabling Tech-*

nologies: Infrastructure for Collaborative Enterprises. Cambridge, Massachusetts: IEEE Computer Society, June 1997, pp. 161–166.

[SC92] P. Strauss and R. Carey, "An object-oriented 3d graphics toolkit," in *Proceedings of 19th ACM SIGGRAPH*. ACM Press, August 1992, pp. 341–349.

[SGM02] C. Szyperski, D. Gruntz, and S. Murer, *Component Software: Beyond Object Oriented Programming*, 2nd ed., ser. Component Software Series. New York, NY: Addison-Wesley Publishing Company, 2002.

[SL98] H. Spencer and D. Lawrence, *Usenet*. O'Reilly and Associates, 1998.

[SSRB00] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann, *Pattern-Oriented Software Architecture Volume 2: Patterns for Concurrent and Networked Objects*. Wiley, 2000.

[Sta00] *SEC 1: Elliptic Curve Cryptography*, 1st ed., Standards for Efficient Cryptography, sep 2000. [Online]. Available: http://www.secg.org/download/aid-385/sec1_final.pdf

[Ste92] N. Stephenson, *Snow Crash*. New York: Bantam Books, Inc, 1992.

[Ste94] W. R. Stevens, *TCP/IP Illustrated*. Addison Wesley Longman, 1994, vol. 1.

[Ste98] R. Stevens, *Unix Network Programming: Networking APIs*. Prentice Hall, 1998, vol. 1.

[Stu96] R. Stuart, *The Design of Virtual Environments*. McGraw-Hill, 1996.

[SZ99] S. Singhal and M. Zyda, *Networked Virtual Environments: Design and Implementation*. Addison-Wesley, 1999.

[Szy98] C. Szyperski, *Component Software: Beyond Object Oriented Programming*, 1st ed., ser. Component Software Series. New York, NY: Addison-Wesley Publishing Company, 1998.

[Tan96] A. S. Tanenbaum, *Computer Networks*, 3rd ed. Prentice Hall, 1996.

[Tre02] D. Treglia, *Game Programming Gems 3*. Charles River Media, Inc., 2002.

[Ult] "Ultima online homepage." [Online]. Available: http://www.uo.com

[Vin95] J. Vince, *Virtual Reality Systems*. Addison-Wesley, 1995.

[vpr06] "Vpr website," Sep. 2006. [Online]. Available: http://www.vrjuggler.org/vapor

[vR06] G. van Rossum, "Python reference manual," *python.org*, no. 2.5, Sep. 2006. [Online]. Available: http://docs.python.org/ref/ref.html

[vrj] "Vr juggler website." [Online]. Available: http://www.vrjuggler.org/

[VRM97a] "Information technology – computer graphics and image processing – the virtual reality modeling language (vrml)," 1997. [Online]. Available: http://www.web3d.org/technicalinfo/specifications/ISO_IEC_14772-All/index.html#International%20Standard%20ISO/IEC%2014772-1:1997

[VRM97b] *Using Spatial Techniques to Decrease Message Passing in a Distributed VE System*. VRML 97, 1997.

[Wat93] A. Watt, *3D Computer Graphics*, 2nd ed. Addison-Wesley, 1993.

[Wat98] K. Watsen, "Bamboo: A portable system for dynamically extensible, networked, real-time, virtual environments," in *Proceedings of the IMAGE 98 Conferences*, Scottsdale, AZ, August 1998.

[WS95] G. R. Wright and W. R. Stevens, *TCP/IP Illustrated*. Addison Wesley Longman, 1995, vol. 2.

[WS98] D. Watts and S. Strogatz, "collective dynamics of 'small-world' networks," *Nature*, vol. 393, pp. 440–442, Jun. 1998.

[WW92] A. Watt and M. Watt, *Advanced animation and rendering techniques*. ACM Press, 1992.

[WW99] A. Wachowski and L. Wachowski, *The Matrix*. Village Roadshow Productions, 1999.

[WZ98] K. Watsen and M. Zyda, "Bamboo – Supporting dynamic protocols for virtual environments," in *Proceedings of the IMAGE 98 Conference*, Scottsdale, Arizona, United States, August 1998, pp. KA1–KA9.

[WZ99] R. Wittmann and M. Zitterbart, *Multicast Communication: Protocols and Applications*. Morgan Kaufmann, 1999.

INDEX