Project Number: RL1-V001

VIRTUTOPIA: A SCALABLE FRAMEWORK FOR ONLINE VIRTUAL ENVIRONMENTS

A Major Qualifying Project Report:

submitted to the faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

in partial fulfillment of the requirements for the

Degree of Bachelor of Science

by

_____

Artur Janc

_____

Matthew Jarmak

_____

Steven Kolk

_____

Robert W Martin

_____

Owen Pedrotti

Date: March 1, 2007

Approved:

_____

Professor Robert W Lindeman, Major Advisor

_____

Professor George T Heineman, Co-Advisor

Abstract for the Final Report of Virtutopia: A Scalable Framework for Online Virtual Environments

By Artur Janc, Matthew Jarmak, Steven Kolk, Robert W Martin, Owen Pedrotti

Virtutopia is a framework for the design and maintenance of persistent virtual online worlds. In this project we designed, implemented and evaluated a proposed architecture for the first stage of Virtutopia.

This report presents the initial architecture, outlining the overall system design, as well as the break-down of the server and client subsystems. The server-side modules include Authentication, Communication, Object Type Repository, Runtime State and Resource module. The client consists of a 3D engine, Engine Interface, Runtime State, Event Manager, Update Manager, Behaviors, Physics and Communication modules. All client and server modules are described, along with the rationale for their inclusion in the system and detailed functionalities.

For each module in the system possible interactions with other modules are presented. Special consideration is given to module parameters and algorithms that affect the overall system performance and the implications of default parameter values in the current implementation are discussed. System-level inter-module communication protocols are presented.

The results of system-level and per-module performance measurements are shown, including server and client stress tests. It was found that under tested load conditions, a single PC-based server can support between 250-2000 concurrent clients, depending on in-game player location.

Table of Contents

List of Figures

1. Introduction

Imagine a long hallway, with doors along the walls. Open a door in the hallway, and you take a peek into an interactive fantasy world, containing dragons and magic. Perhaps you choose another door, and you join a team of bioengineers designing a new painkiller medicine, or you open yet another door, and enter a fast-paced death match shooter game, where your survival depends solely on the speed of your reflexes. The image in your mind is what the essence of Virtutopia is: a gateway into many various virtual environments where thousands of users can interact with artwork, solve engineering problems, or lose themselves in an online video game.

Over the past 10 years, the computer gaming industry has seen the rise of the Massively Multiplayer Online (MMO) Role-Playing Game (RPG) genre, which consists of games like Everquest [6] and World of Warcraft [7]. The key aspects of this genre are the ability for people all over the world to connect to one central "world", or virtual environment, where the people can interact cooperatively or competitively to work their way through the goals of the game. These types of games have become immensely popular, with a user base that numbers in the millions [2].

The “massively multiplayer” concept has many more applications than just role-playing games. First-person shooters (FPS), real-time strategy (RTS) games, as well as educational and academic software, such as artistic collaboration or interactive engineering design, can all benefit from an underlying multi-user framework.

Virtutopia aims to facilitate the process of development of such multi-user environments by providing a scalable and modular application programming interface (API), which will allow programmers and content creators to design their own virtual worlds.

2. Project Objectives

When it is finally completed, Virtutopia will be a scalable distributed networked virtual environment, usable in massively multiplayer games, as well as other virtual simulations. The specification for this first stage of the Virtutopia project, however, is to create a scalable framework supporting games and virtual environments, as well as simple proof-of concept applications to demonstrate its functionality.

This framework will include a protocol for client-server and peer-to-peer communication within the system, as well as a way to property-rich objects and transfer them over the network. The client program built as part of this MQP will be platform independent, using the C4 3D rendering engine [5] for the user interface. The server architecture will be scalable through distributed systems using the QNX Neutrino real-time operating system [4].

Also, this project will result in a massively scalable communications framework on top of which an interactive computer game or a virtual environment can be created. By not assuming any specific genre of game when designing the system, we will preserve the ability of this framework to be used for games of many different genres.

Considering the long term goals for the Virtutopia environment, another important objective for this project is to create a modular, scalable and robust framework design, adhering to industry-standard software development methodologies. Necessary components of such a design include extensive sets of unit tests, integration tests and multiple levels of documentation ranging from high level architecture specifications to implementation-specific guidelines and comments.

3. Theory

As the scope of the Virtutopia project is broad, this MQP team will complete the design of the system, without implementing some of the functionality desired in the long term. The main implementation deliverable for the current project is to create a functional scalable server subsystem on the QNX platform, which will allow a user to log in and interact with a Virtutopia world (*realm*). To fulfill this requirement the implementation of a basic working client will be necessary.

An important performance goal for the system is to significantly increase the industry limit of possible user connections for a single game server. The current state of the industry is for a single server in an MMO game to support approximately 2,000 users [1]. The number of users of World of Warcraft in Europe is 1 million. Taking into account that at any given time about 20% of that user base is actively connected to the system, there are approximately 200,000 users actively connected to World of Warcraft in Europe. According to the World of Warcraft official website [3] there are 95 European servers, which indicate an average of 2,105 users per server. This indicates that the expected number of players handled by a single server can be estimated at about 2,000.

The definition of a "user" in this case is an entity that requires successive updates on the game state from the server. In our architecture, each entity that the server actively transmits game state data to is an entire peer-to-peer group (referred to as a *peer group* for simplicity). Thus the theoretical maximum number of users per server on our system is 2,000 * (users per peer group). According to Chen [1], "Older hardware can handle low-latency shooters with 32 or more players, suggesting that modern systems running MMO games, with their significantly relaxed performance requirements, should manage many dozens of players without problems." For the purposes of this estimate, we interpret "many dozens" to be 96. Therefore, for a typical MMO game running on our architecture, the best-case maximum number of users that can be simultaneously connected to one server is 192,000. This represents a gain in user support of two orders of magnitude.

In contrast to the characteristics of RPGs, FPSs typically require far greater bandwidth. The state of the industry for a typical low-latency FPS is 32 users per server [1], supported on a typical high-end PC. A "user" in our architecture represents a peer group, which theoretically can contain up to 32 users [1]. Taking into account that there exist network games using the P2P network architecture supporting up to eight users per game (such as a popular real-time strategy game – Starcraft [8]), we can expect to form peer groups with about four users. This will account for the fact that a 3D game will have higher bandwidth requirements, in addition to the larger amount of bandwidth needed by the peer group leader.

Therefore, for a low-latency 3D game utilizing our architecture, on a commodity PC, the expected number of users per server is 4*32 or 128. The explanation for why this is so drastically lower than our estimated 192,000 users in an MMORPG is two-fold. First, the bandwidth requirements for a FPS game are much higher than an RPG, because the genre of game requires updates to be sent out more rapidly. Also, the number 128 reflects expected performance on a commodity PC, which we are using for our server. The number 192,000 reflected an RPG game running on a commercial-quality server cluster, which has far greater computing and network resources.

4. Comparative Product Survey

The outcome of the Virtutopia project is a framework for interactive persistent virtual environments. There are several technologies and approaches for building environments that would allow users to communicate and interact in some sort of a virtual world. In this chapter we present a few of these approaches.

The most widespread architecture for MMO games today is a multi-tiered client-server architecture. When a client machine wishes to connect to a game session, it first connects to a master server (which can be a single server, a cluster of servers, or a datacenter), which handles the user's password and login authentication. The master server then connects the client machine to one of many game servers [1].

Each of these game servers can host up to approximately 2,000 clients simultaneously. While 2,000 clients connected to a single game server may be a significantly large number, it is still a small fraction of the millions of users that play the more popular MMO games, like *World of Warcraft*, or *Everquest*. As shown in Figure 4-1, in order to accommodate the massive numbers of users connected to the system, most modern MMOs use a technique called *instancing* [1]. The entire game system is comprised of many game servers, each of which hosts approximately 2,000 clients, and each of which is managing a separate, isolated copy, or instance, of the game world. So, while the "virtual world" of the MMO game contains tens of thousands, possibly even millions of users, no single user can interact with every other user. In reality, the MMO game has many isolated copies of a "virtual world," each of which only contains approximately 2,000 users [1]. Any changes to one instance are not seen in any of the other instances.

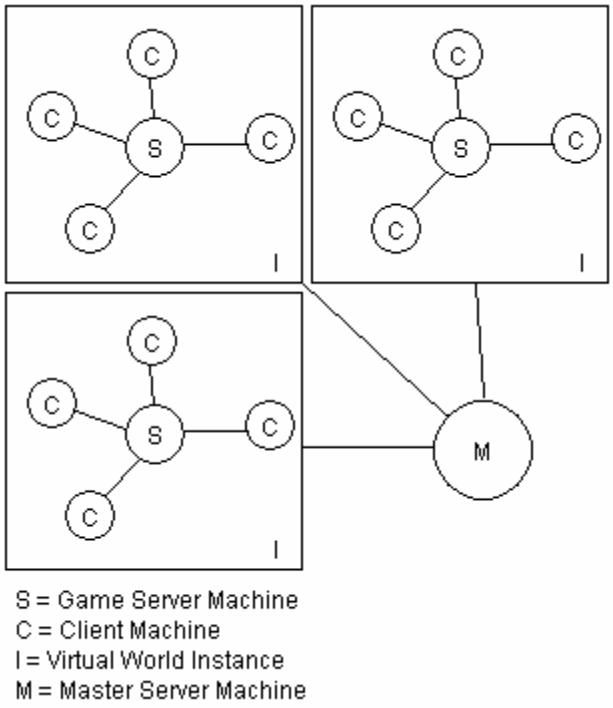

Figure 4-1: Client-server Architecture with Instancing

As with any system design, there are both advantages and drawbacks of the above system. One of the major advantages of this system is that all of the security and authentication policies used by the system are maintained and mediated by a central point that is under control of the game's publishers/developers. Another advantage is that the concept is rather straightforward, and relatively uncomplicated to develop. Finally, if the game's publishers decide that more instances of the game are needed, then they can be added into the system without interrupting the other server instances that are running [1].

This system also has several disadvantages. The expense of purchasing and maintaining these servers is a major limiting factor for most publishers. Also, the robustness of each game instance is limited, due to the fact that each instance has a single point of failure. The master server is also a single point of failure which can bring down the entire system. Perhaps the most significant limitation of this system is that the virtual world is only partially scalable, because even though more users can be added into the system via additional instances, the entire set of users cannot interact with each other across instances [1].

An alternative approach proposed by Alvin Yun-Wen Chen introduces the concept of using a hybrid Client-Server and Peer-to-Peer approach [1]. As in the above architecture, when a client machine initially connects to a game session, there is a master server that handles all of the required security and authentication of the user. However, for the rest of the game session, the architecture is drastically different, and leverages some common aspects of an MMO game [1].

In many modern MMO games, the style of game play is such that small groups of users, usually around 20 or fewer, tend to form small teams, or "parties" of players that travel around inside the virtual world, and either competitively or cooperatively play through the game. If such a group were playing in a game with a traditional client-server architecture, then the game server would be responsible for mediating all of the game actions. The hybrid architecture, however (depicted in Figure 4-2), would delegate responsibility for the game state in a specific region to a small peer group of players. The peer group would occasionally send the server updates to the game state so that the server can continue to manage game state persistence. Whenever the number of players in a certain group exceeds a practical number, the peer group then returns control to the server, so that its increased system performance can mediate the now much larger group of clients [1]. One client in each peer group is selected to be the single point of contact (POC) between the peer group and the server. POC status might transfer to another peer within a group for a number of reasons, such as the POC leaving or the server suspecting the POC of cheating. In this way, the entire peer group appears to the server as a single client, thereby increasing scalability.

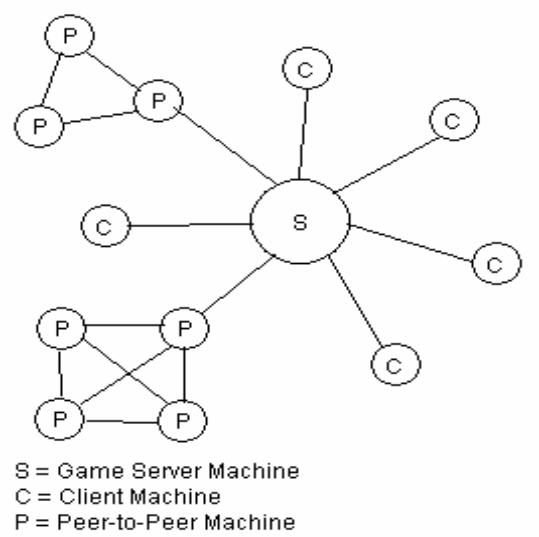

Figure 4-2: Hybrid Client-server/Peer-to-Peer Architecture

The advantages of this system architecture over the straight client-server model are many. Firstly, because the vast majority of players tend to go off and form small groups to play the game together, many of the clients connected to the system are processing the game state and mediating the game themselves, which allows network bandwidth and processing power of the server to be saved. If the server has to do less work to maintain the same number of users, then the system can scale to a larger number of users. This architecture converts the massive scale of the game world from a limitation into an asset; the processing of the game state is offloaded to the users currently connected to the system. Secondly, the cost of maintaining the servers is decreased because the same number of clients can be managed by fewer servers. The reduction in server load could also allow virtual worlds to increase in complexity [1].

There are drawbacks to this system architecture, however. One such drawback is that the system is much more complex than the plain client-server architecture, meaning that development is more difficult, and therefore will likely take longer to complete. Secondly, because the game's client programs now handle a heavier workload that includes some of the server's old responsibilities, security can be more difficult to mediate because a compromised client can now be managing the game state. Finally, there is a significant increase in the workload of the client machines. This comes from the fact that each client in a peer group needs to send world state updates to the other peers in

his group. In addition to this, the client must send updates to the server whenever there are multiple peer groups in a small region of the game world, and also receive updates about the other nearby peer groups from the server. This increased workload that is placed upon the client machines creates the need for the game's customers to own more-powerful machines in order to meet the requirements for the game. This may cause the game to lose a portion of its user base because it does not support older machines [1].

5. Project Specifications

The primary deliverables that will be created by the completion of this project are the server program, the client program, and the engine interface program. These three programs fully represent a working version of the Virtutopia framework that users can connect to.

The server program runs on the QNX Neutrino Real-Time Operating System. In theory, the server could support a large number of simultaneous users (see theory considerations in section 3). However, we are running the server on 3.2 GHz Pentium 4 machine with 512MB of RAM, reducing the number of simultaneous users significantly from our theoretical maximum. The reason for this is that our server machine has less computing power than the typical professional game server. As such, we plan to support 128 simultaneous users for our final demo. However, the number of users that we can support could theoretically grow by increasing the power of the server machine to a professional game server, such as one used by Blizzard, Inc. for their MMO game, *World of Warcraft*.

The client program will run on Microsoft Windows 2000/XP/Vista and will interface with an instance of the game engine running locally, so that it can offer meaningful audio and visual output to the end-user. The client program will be tested on machines of various computing power, to ensure that the program does not need a specific computer to run. Two hardware configurations which will be tested are a 2.8GHz Pentium 4 with 512MB of RAM and an NVIDIA GeForce 6200 video card, as well a machine with a 2.8GHz Pentium D with 1024MB of RAM and an NVIDIA GeForce 7300 video card. The potential also exists to test the client running on a machine with an Intel Core 2 Duo E6600 processor with 2048MB of RAM and an NVIDIA GeForce 8800GTS video card. The engine interface program is written on top of the C4 Game Engine, and will interface with the client program for the purpose of returning user input to the client as well as delivering visual output to the end-user. This Engine interface program only has a Windows 2000/XP/Vista version, because there is no known version of C4 that runs on Linux; while all implemented modules should compile for Mac OS X, due to the lack of

hardware and a development environment for this platform we did not attempt to build the Macintosh client. However, the client program can be run in faceless mode, meaning that the client will not be connected to any engine. This mode will allows testing the Linux version of the client application without requiring that C4 runs on the Linux platform.

While we have chosen the C4 engine for our final demonstration of the Virtutopia framework, the engine runs in a separate process from the client, so that an engine interface can be easily written using any game engine without having to modify the Virtutopia client program itself.

6. Virtutopia System Design

For Virtutopia to be a successful framework for networking distributed virtual environments, several key design principles and development concepts must be adhered to throughout the entire design and implementation process. An important aspect is that many different types of games and in-world interactions will be possible. The Virtutopia framework must be ignorant of the particular game-type to enable the system to support various uses, including ones not specifically considered by the designers.

Since Virtutopia realms will provide services to many users connected concurrently, the framework must also be scalable, so that truly distributed and massively multiplayer online interactions are possible.

6.1 Overall System Architecture

The Virtutopia system architecture is based on a hybrid client-server and peer-to-peer approach [1]. When a player uses a client to connect to the server, the client logs into the virtual world hosted on that server, and the state of the player's character and its surrounding environment is loaded from the server to the client. The player can then enter the virtual world, and interact with the world and the other participating players.

Eventually in the course of the game two or more players will meet in the virtual world. When this happens, the server will arrange those players into a peer group which handles each player's interactions with its peers and the game world. Possible interactions include moving within a room, creating objects, and chatting with other players.

The peer group will periodically update the server's knowledge of the game state in the region being managed by sending an update message to the server's Runtime State module. When two peer groups encounter each other in the game world, the server will handle reconnecting them to form a larger peer group that controls the game state in the region they are occupying, or, if the number of peers in the two groups is larger than the

maximum allowed peer group size, the server will forward data between both groups, allowing clients in one group to interact with the other.

If the number of peers in both groups allows for the merging of peer groups, the server will invoke its liaison-assignment algorithm to determine the liaison of the new group.

6.2 Server-Side Architecture

The overall block-level design of the Virtutopia server application can be seen below (Figure 6.2-1).

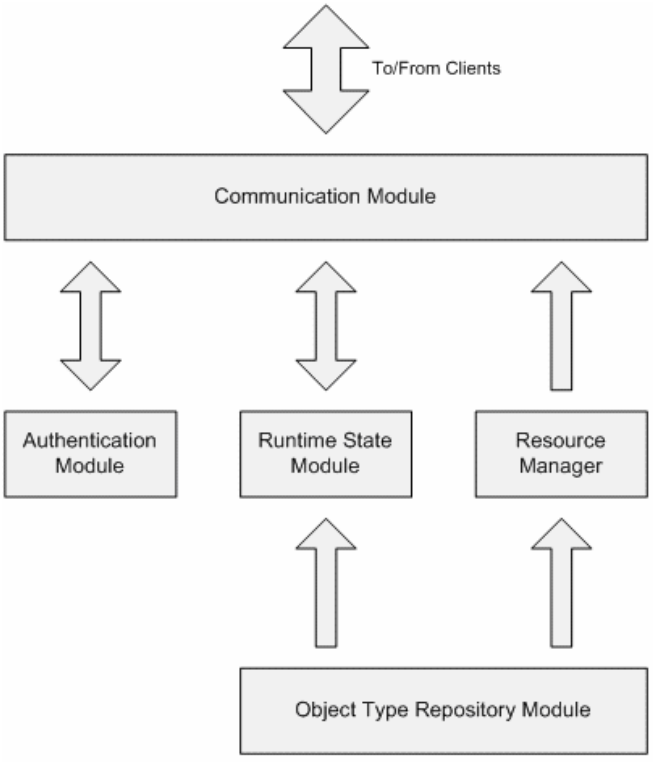

Figure 6.2-1: Virtutopia Server Architecture

The server application has been designed to run on the QNX Neutrino Real-Time Operating System (RTOS). One of the many useful features of the QNX Neutrino RTOS is that an application written for this environment consists of many small processes that communicate with each other through the use of Message-Passing. Message Passing in QNX is an extremely efficient form of Inter-Process Communication (IPC) that allows data to be sent from one process to another without needing to context-switch into kernel space. By avoiding context-switching, QNX Message-Passing minimizes overhead.

The advantage of using multiple processes that communicate with each other using QNX messages is that the functionality of the server can increase through the addition of new lightweight modules. This improves the maintainability of the code-base, since functionality is separated into multiple interacting processes, rather than one large monolithic system. It also allows designers to create modules which can be swapped in and out of the system, without affecting other running modules.

It is important to note here that when we make references to the term “module” on the server-side, we are referring to a small process that other modules can interface with and communicate with via QNX Message Passing.

6.3 Client-Side Architecture

The overall block-level design of the Virtutopia client application can be seen below (Figure 6.3-1).

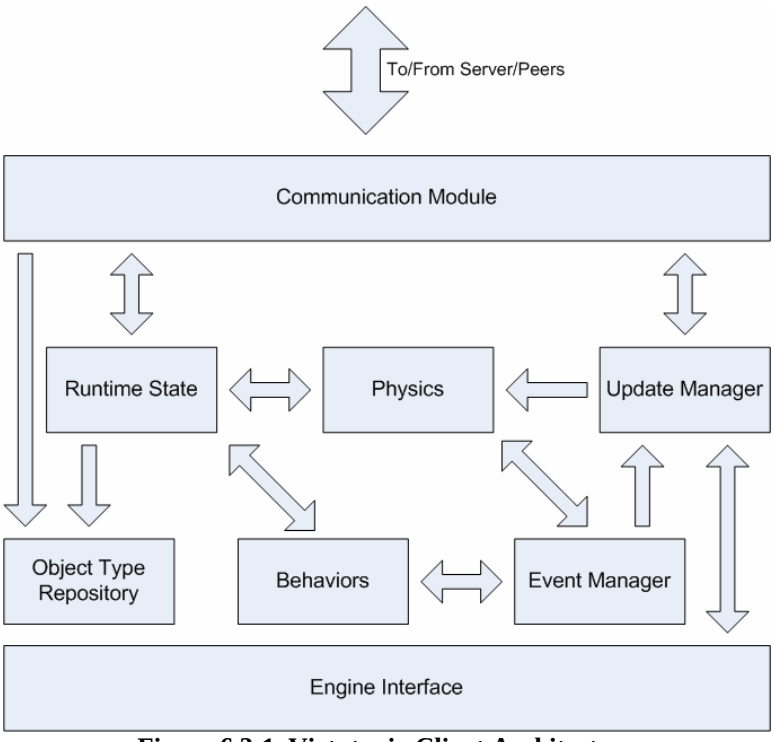

Figure 6.3-1. Virtutopia Client Architecture

The client application is the system run on an end-user’s personal computer. Its primary purpose is to provide the user with an interactive interface into the virtual world, as well

as to assist other clients and peers in the system with the processing of the simulated world. The client application has so far been implemented for Microsoft Windows (2000, XP, Vista), but all modules are compatible with Mac OS X. The client will also compile on the Linux operating system, although with no 3D display capabilities.

Cross-platform compatibility has been achieved through two means. First, the 3D rendering engine that our client utilizes, C4, is fully compatible with recent Microsoft operating systems, as well as the Mac. Second, the multithreading and networking libraries used to implement the system are based on the POSIX standard, and therefore are also cross-platform.

Unlike the server application, the client is composed from just two separate processes: the client itself and the rendering engine. The reason why the client application itself is in one large process is that IPC on Windows systems is less efficient than on QNX Neutrino systems. Also, the reason for the client/engine segregation is that in the future, multiple different engines could potentially act as the renderer for a Virtutopia client. Also, a particular client may wish to join the simulation to aid in the workload distribution, while having no interest on the rendered output on a display. In this case, the client can simply connect to the server, but not start up the engine process.

It is important to note here that when we make references to the term “module” on the client-side, we are referring not to a process, but rather a singleton class that other modules can interface with and communicate with via standard C++ function calls.

6.4 System Module Architectures

6.4.1 Authentication Module

The Authentication Module is responsible for ensuring that each player using the client to interact with the virtual world is a valid user of the system. This module maintains a database of all the user accounts that are currently registered in the system.

Upon receiving a login request from a client, the Communications module on the server queries the Authentication module to determine if a username and password pair is linked

to a user account in the system. The Authentication module queries its database to search for the identified user account. If it finds a match, it responds positively, enabling the connected user to participate in the Virtutopia simulation. If the Authentication module cannot find a match, then the connection with the client is terminated because of invalid login credentials. As with all server modules, the Authentication module is its own small process that is communicated with through the use of QNX Message Passing.

The original design for the Authentication Module called for an SQL table to store the user accounts. This table was configured using MySQL on the QNX platform. It contained a unique integer identifier for each user account, along with a username and password field for each account. However, during the implementation of the server application, we discovered several bugs in the QNX implementation of MySQL. These bugs caused our code to malfunction, making debugging our entire system very difficult. The MySQL code that we used had been tested rigorously on Linux and UNIX systems, so we knew that the problem was not with our code. As a workaround, to keep our development process moving forward, we switched to storing the user account data in a plain text file. This implementation was sufficient for our purposes. The MySQL implementation is still there, it is just not being used currently.

6.4.2 Object Type Repository Module

For a virtual world to be believable, it must contain a rich set of objects. As such, any virtual world needs a flexible and extensible way to represent the objects that populate the world. In the Virtutopia framework, the representation of in-game (or in-world) objects is split into two main components.

First, we introduce the concept of the Object Type Repository. The server's Object Type Repository (OTR) is a monolithic storage facility for every object type in the world. These object types are immutable while the system is running. An object type is a concept such as "chair", "donkey", or "pencil". Each type of object in the OTR is represented by a unique identifier string, made up of alphanumeric characters. The format of this identifier string is "a.b.c.d$", where each entity between periods is a word, like

“world” or “object”. One such example of this is “world.object.chair$”. The OTR was originally designed to use an SQL database to store these unique strings as well as the file system location of an XML file that contains the properties that describe each object type. The reason why the specific properties of objects would not be included in the SQL table is that for any specific game, the properties that an object has may be different. For instance, in a role-playing game, a property like “gold cost” would determine how expensive items are in shops, whereas in a first-person shooter, a specific firearm might store the number of bullets in its magazine. The XML fragment associated with a type-specific property will be formatted as `"<prop type=int> <name>foo</name> <value>11</value></prop>"`. In order to keep the OTR flexible and extensible, we’ve opted to store the object type-specific properties in XML files, and have the SQL table store every property an object *must* have, such as its unique identifier, and the location of its XML file. However, as development of this module progressed, it became apparent that the SQL tables were not needed for the OTR. This is due to the fact that there exists a one-to-one mapping from an Object Type’s unique identifier, and the path for its XML file. The mapping is quite simple; replace all ‘.’ and ‘$’ characters in the identifier with ‘/’ and then append object.xml. This mapping is convenient because it allows each object’s XML to be stored in a unique and intuitive location. Also, checking whether an Object Identifier is valid simply becomes a file lookup. If the filename exists, then the Object Type exists; if it doesn’t, then the identifier is invalid. In addition to the properties of an Object Type, the XML file also contains file system links to the audio and graphical information that an object type has, such as texture maps, 3D meshes, and sound effects. These are stored in separate files from the XML because binary data is difficult to store within a text-based file format such as XML. On the server-side, the OTR module is its own separate process that communicates with other modules through the use of QNX Message Passing. The data structure that contains the Server-side OTR is a Trie data structure. Trie data structures are more efficient at searching than binary trees when the indices of the tree are character strings.

On the client side, there exists a smaller version of the OTR, called an object “reposlet”. This storage facility only stores a small subset of the server’s OTR, because each client

only needs to know about the object types that it can interact with in the region where the player exists. This reposlet stores the same type of information that the server stores, but it can store it in working memory rather than in XML files. We don't want the client side to rely on SQL tables because we do not want the end-user to need a version of SQL running on their personal machine. Also, if the client's reposlet is stored on the hard drive of the user's personal computer, then there is a security risk in the sense that a user might be able to edit some of the data that represents the objects in the reposlet. However, if the data is stored in the working memory of the user's system, then it becomes more difficult for the end-user to edit or otherwise modify the game's content. To achieve this, the data structure that contains the Client-side cache of the OTR is a Trie data structure, just as with the Server. On the client-side, the OTR module is represented by a singleton class that other modules can access through standard C++ method calls.

6.4.3 Runtime State Module

While the OTR on the server and the reposlet on the clients are excellent at handling types of objects, we still need a way to instantiate these objects into the game world. Thus, we introduce the concept of the server's runtime World State (WS). The WS stores each instance of any object that currently exists in the game world. In addition, the WS can instantiate new objects into the world and remove objects from the world. A practical example of this functionality is a room that has five balls bouncing around inside of it. The WS would know that inside this world, there are the following objects: five balls, four walls, a ceiling, and a floor. The OTR knows about the following object types: ball, wall, floor, and ceiling. Every object that currently occupies the world has a unique identifier that allows this single object to be distinguished from other object instances. This identifier is a 64-bit unsigned integer. Each of these "objects" is actually the instantiation of an object type that is stored in the OTR. The WS stores the unique identifier string that identifies what type of object it is. Also, the WS stores what the current values of the object instance's properties are, such as its current location, velocity, whether or not it is hungry, or whatever information the game needs to store. On the server-side, the WS (RTS) module is its own separate process that communicates with other modules through the use of QNX Message Passing.

On the client-side, there exists a smaller version of the WS, called the runtime Region State (RS). The RS has the same responsibilities as the server's WS, except that it only contains information about the objects in the region of the game world that the player currently occupies. The reason for this is simple: if there existed a virtual simulation of the WPI campus, a player running around in Fuller Labs would not need to know how many desks are inside Olin Hall. Since the player has no need for that information, there is no reason to pull it down from the server and store it on the player's personal computer until that player enters Olin Hall himself. As with the server WS, the client's RS is capable of deleting objects. When this occurs, a network message is sent to the server so that the deletion occurs in the server's persistent WS. On the client-side, the RS (RTS) module is represented by a singleton class that other modules can access through standard C++ method calls. Both the client-side RS and the server-side WS will store a copy of the Runtime State in memory, because it is too slow to read it from disk every time another operation needs the state information.

6.4.4 Resource Module

The Resource Module exists solely on the server-side of the system. Its responsibilities include forwarding game or simulation resources to the Communications Module so that they can be forwarded to the clients who request said resources. Resources that the client would be interested in include XML files to describe Object types, Lua script files that describe an object's event behavior, 3D meshes and texture images for graphical rendering, as well as sound effects.

The Resource Module receives requests for resource downloads from the Communications Module, fetches the requested resource(s) from the server's hard disk, and sends the data contained in the resource file to the Communications Module, where it is then forwarded to the appropriate client machine. In the case of large binary files, such as sound effects, texture images, and 3D meshes, the Resource Manager forwards a file system link to the Communication Module, and it performs the file I/O to avoid having to copy large quantities of binary data around in system memory. However, for small text-

based files, such as XML object files and Lua script files, the text contained in the file can be directly forwarded to the Communications Module.

The Resource Module is capable of responding to requests for a specific resource, or a request that asks for the entire set of resources needed to run a specific simulation. In the latter case, this module parses through the XML File that describes the Game State, or Simulation State, so that it can gather a comprehensive list of resources that satisfies the client's needs. In addition, other server-side modules can submit 'additions' to the master resource list if they discover that another resource may be needed. On the server-side, the Resource Module is its own separate process that communicates with other modules through the use of QNX message passing.

6.4.5 Communications Module

The purpose of the Virtutopia Communication Module is to provide an abstraction for higher-level client and server modules, allowing each client to communicate with other players in the game, as well as the central server. The network layer manages network connections, provides facilities for encryption and compression of traffic, and ensures reliability of in-game communication.

The Virtutopia network is designed using a hybrid architecture, combining some aspects of a client-server model with a peer-to-peer environment. In the virtual world the server arranges users into *peer groups* based on pre-defined selection criteria, such as the time of connection, or location in the Virtutopia environment. An example of a Virtutopia network topology is shown in Figure 6.4.5-1.

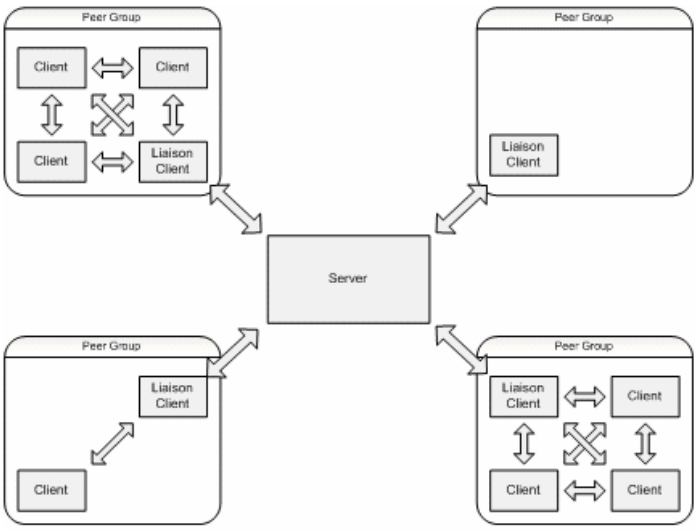

Figure 6.4.5-1: Virtutopia Network Architecture

The hybrid approach was chosen as a compromise between server scalability and maintaining a synchronized central state in the global server database. Depending on the virtual environment created with Virtutopia, the server may or may not need to synchronize and police all game state changes.

If server synchronization is necessary, the hybrid model can fall back to a traditional client-server approach if the peer group size is set to 1. Otherwise, the server can treat each peer group as one game client, utilizing only as much network, memory and processing resources as one traditional client-server connection. In an ideal case, the server would be able to connect as many peer groups as it could support single clients with a traditional approach. Realistically, there will be overhead incurred by the need to manage peer groups, as well as handle group formation and deletion, limiting the number of peer groups which can be handled by the system.

Design goals

One of the most important goals of the entire Virtutopia framework is to allow developers building virtual worlds to choose appropriate algorithms and approaches specific to their application. The Virtutopia network design mirrors this objective by providing different policies and mechanisms which implement them, so that developers can choose the behavior most appropriate for their system.

Another important goal of the network design is to abstract away all network-related functions to developers, so that they can focus on the high-level aspects of the application knowing that necessary game state updates and other messages are seamlessly and reliably received.

Most of the currently available massively multiplayer environments require significant hardware resources on the server side to allow reliable connection and game state management. The Virtutopia network model is designed to increase server scalability by reducing the amount data the server must process through delegating some of the processing tasks to clients.

6.4.6 Physics Module

Without physics, there is no movement, no collisions, and in general little interactive output. The responsibility of the physics module is to handle such tasks. It deals with Newtonian kinematics to apply the effects of gravity, air resistance, and other universal forces. (In the current release of Virtutopia, gravity is the only universal force implemented). It is important to note that it only deals with universal forces - that is, forces that affect all objects equally. If a force does not affect all objects equally, then it is the domain of the Behaviors module, not the Physics module (described in Section 6.4.7). The Physics module works closely with the Behaviors module to perform collision detection. As each object has a different shape, it also has a different collision volume. This collision volume is represented as a polymorphic object, such that the Physics module does not care what the details of the collision volumes are; it simply tells them to check if they are colliding. They themselves know what their own collision algorithms are.

6.4.7 Behaviors Module

Not every object reacts in the same way to the same stimuli. The Behaviors module relies on Lua scripting to provide a dynamic response to events, regardless of what the event actually is. The Behaviors module also performs the collision algorithms for objects, after

retrieving their collision volume from the Runtime State. It is helpful to remember that by default, every object will react in some way to a collision, so this is the one event for which a script should always be invoked. Of course, if an object should do nothing when it collides (that is, pass through other objects as if it were not there), then an empty script should be provided for the object.

Whenever an event is processed, Behaviors checks if an object being acted on in the event has a script for that event. If it does, it executes the script, after giving it the data it needs from the event. In this way any number of event types can be dynamically added and supported without need to recompile the core code. It is also useful to note that all Lua code is encapsulated within a wrapper class. Should a later implementation decide to change the scripting language used, only that class will need to be changed.

6.4.8 Engine Interface Module

The Engine Interface is broken into two parts: the client module, and the engine application. These parts reside in different processes on the client machine and use sockets to communicate with each other. The client module, as its name implies, resides inside the main Virtutopia client, and will not change as different engines are used. The engine application, on the other hand, is completely independent of the client and will vary widely in its implementation for different game engines. This creates a decoupling of the general Virtutopia client code and the specific engine code which allows developers to easily adapt their own game engines to the Virtutopia system.

The two primary purposes of the Engine Interface are to receive user input and to render the scene. This is accomplished by passing messages back and forth between client and engine. The update manager, described below, is responsible for sending updates to the interface, and for polling the interface for user input messages through the Engine Communication object. Engine Communication will then handle the task of sending and receiving data. Engine Communication is also responsible for starting the engine process itself when it is told to do so by the client. It will run the application, and then connect to it once it is available.

The engine application consists primarily of glue code, as well as a mirrored version of the Engine Communication module called Client Communication. Client Communication handles sending and receiving data from the Client, structuring the data in such a way that it can be understood by the engine itself. The glue code then takes that data and feeds it to the C4 engine, overriding C4's own physics and collision detection to simply display objects on the screen without extraneous processing. More glue code then collects user input data from C4 and sends it back to Client Communication. The Client Communication module itself would not need to change from engine to engine; though it is not strictly necessary that it remain the same, so long as engine communication protocols are implemented correctly. Because the communications consist only of string based socket messages, anyone could write their own engine application, in any language that supports sockets.

6.4.9 Update Manager

The Update Management module is an integral module of the Client architecture. It is what starts, processes, and completes every time slice of the Client. A time slice is a discrete period of time with variable length that is specifically started and ended by the Update Manager. It coordinates and synchronizes all of the other modules to produce the output of every time slice.

Processing a time slice is a five step process that involves calling several modules (Figure 6.4.9-1).

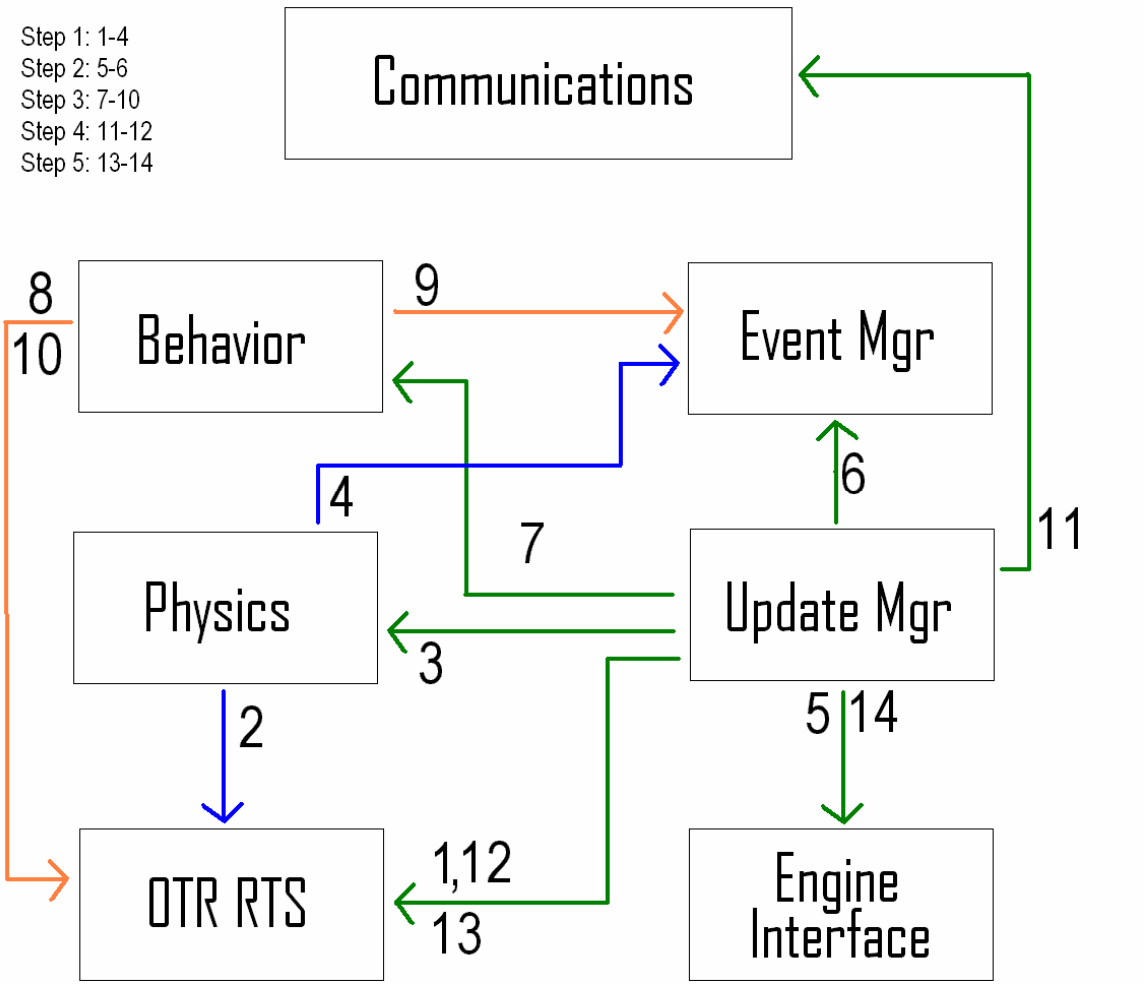

Figure 6.4.9-1: Processing a time slice.

1. **Update** – All objects in the runtime state are updated. The Update Manager calls the Physics module and requests that it update each object. After Physics has completed updating an object, the Update Manager then checks the amount of time elapsed since last sending a network update. The Update Manager contains a specific interval pertinent to sending network updates. When the amount of time since the last network update was sent has exceeded this interval, the Update Manager will generate an update for the current object and send it to the Communications module for transfer to the client's peers. If the Update Manager processes time slices slower than the network interval, it will send out a network update every time slice. This is good behavior because the network interval is the fastest speed at which updates should be sent. Updates cannot be sent faster than

they are processed, so by using an interval it allows the Update Manager to send updates as fast as it can, using the interval as the maximum.

2. **Process Input** – All input from the user is processed by calling the Engine Communication module, which relays any input that was received from the engine. Any received input is placed into an event and sent to the Event Manager.
3. **Process Events** – The event queue represents the events that should occur in this time slice. To evaluate an event from the queue, the Update Manager sends the event to the Behaviors module. The Behaviors module handles calling scripts or writing data to the runtime state. This is repeated until the event queue is empty. This step is a potential bottleneck as the number of events escalates with the number of objects. It was for this reason that a potential solution, only processing events up to a certain amount of time, was considered. However, it was realized that if the amount of time to process the events was consistently longer than the set amount of time, the runtime state would begin to lag behind what the state should be. It was for this reason that it was decided to evaluate the entire queue every frame instead of stopping after a certain period of time.
4. **Process network updates** – All network updates received during the time slice are retrieved by accessing a queue within the Communications Module. While the Update Manager accesses the update queue, the queue is locked using a mutex to prevent it from increasing in size while it is being processed. Each update is taken from the queue and then written to the runtime state. This is repeated until the update queue is empty. The mutex is then released so the queue can be added to again.
5. **Update engine** – The engine is updated by sending each object from the runtime state to the Engine Communications module.

This process of generating a time slice repeats many times a second to produce the final product of an interactive world with physics and behaviors.

6.4.10 Event Manager

The Event Manager Module is responsible for the creation, storing, and getting of events that occur in a time slice. An event represents either a change to a single object in the runtime state or a creation of more events. This includes, but is not limited to, changing an object's position, user input, and an object colliding with another object. It stores the events in a linked list queue, with pushes going onto the tail of the queue and pops coming from the head of the queue. The queue is added to by the Behaviors, Physics, and Update Management modules and evaluated and written to the runtime state by the Update Management module.

6.5 System Communication Protocols

6.5.1 Network Protocols

Connection Management

The Virtutopia network functionality is provided by the client and server Communication modules. On both the client and the server the Communication module provides network communication services to other modules, abstracting lower level and system-specific implementation details.

The primary responsibilities of the Communication module are to listen for connections from other clients, send messages passed to the module by higher level Virtutopia layers, and receive and appropriately dispatch incoming messages.

The Connection Management sub-module is responsible for keeping track of all clients which have open network connections to the server at any given time. This sub-module can also deny service to new clients if the maximum connection limit for the server has been reached (this limit can be determined based on the resource utilization of a specific game, and modified in the server configuration). If a user gets disconnected from the server, this sub-module will detect that fact, update the list of connected clients, and ensure that the user will be able to connect to the system again after he/she regains the connection.

Connecting to the Server

A connection to the Virtutopia server is initialized whenever the user types in the server hostname into a connection dialog in the user interface. The user will also input his username and password, which will be put into an authentication message and sent to the server. If the credentials are valid, the server will issue a login confirmation and add the user to its list of all connected users.

The client will then ask the server for the server game time, so that all players will have a synchronized game time. When a reply is received, the server time will be saved, and a difference between client time and the received server time will be stored and applied to timestamps for messages sent to the client's peers.
After the time has been synchronized, the client will proceed to retrieve the state of the region in which the user's character is located.

The client will then query the server for the peer group to which it was assigned. Upon the receipt of a reply, it will connect to all the peers in its group and start receiving game state updates.

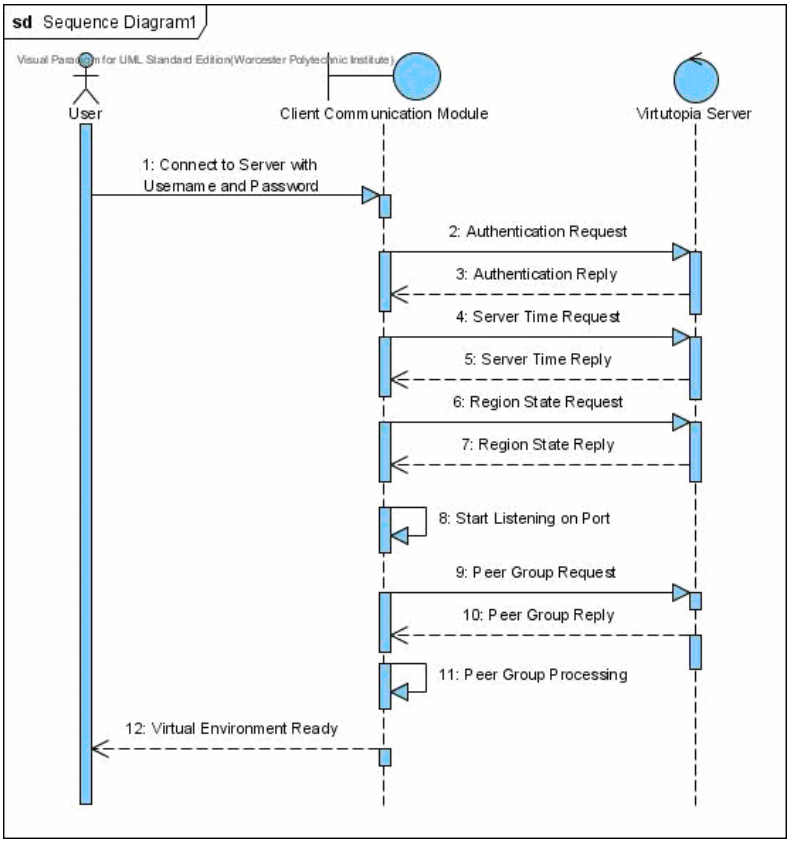

Figure 6.5.1-1: Virtutopia Connection Process

Parameters and Tradeoffs

1. Persistent connection to server. Currently: enabled.

Each client might maintain an open network socket for communicating with the server. This will allow the server to more quickly and efficiently send messages to all clients, but will consume server resources by allowing a large number of persistent open sockets.

Peer-to-Peer Management

In each peer group, users communicate directly to one another, sharing their characters' location and actions. Client messages are sent only to other peer group members. In every peer group there is a designated *liaison* – a client who, in addition to sending and receiving messages from other peers, at a specified interval informs the server of the changes in virtual world state caused by all clients belonging to the peer group. The update interval can be specified by the developer as a parameter for the Communication module. The current Virtutopia implementation uses the same value for liaison-server

updates as for state updates within the peer group; thus, the server always receives current state information consistent with the state of each client in the peer group.

The hybrid architecture can be made to fall back to a regular client-server approach by selecting a maximum peer group size of 1. In this case each client will perform the function of a liaison, periodically communicating directly with the server to inform it of modifications it has made to the world state. This approach allows the server to directly oversee and police the actions of each client, and enforces direct server control over the game world state. However, it also eliminates the performance benefits of splitting users into peer groups, requiring the server to utilize more per-user resources and reducing scalability.

The server's knowledge of the state of the world is therefore not always completely accurate. A peer group might make changes in the state of a certain region, and until the liaison sends an update, this state will be local, and will not be reflected in Virtutopia server's database. At any time the server has an ability to request the state from each peer group by sending a message to its liaison.

Assigning users to peer groups.

The process for forming user peer groups is crucial for the functioning of the hybrid network architecture in Virtutopia. The choice of algorithm for determining which peer group a connecting user should be assigned to is non-trivial and can be dependent on the goals of the system utilizing the Virtutopia framework.

A natural choice is to group users based on their in-game location, so that each group can control a specific virtual area. For some types of applications, where low latency is of significant importance, there exists a possibility of grouping users based on their network proximity, even when their locations in the virtual world span a large area.

The current algorithm is to assign users to peer groups based on the order of connection to the server. This approach has been chosen for the initial implementation stage, as it is

simple to develop as well as game-location agnostic, and does not require any kind of partitioning of the game world.

Liaison Assignment

The algorithm for choosing the client to become a liaison for a particular peer group requires careful consideration. An important realization is that the liaison has the highest potential for in-game abuse, as it is the only client within a peer group which directly communicates with the server, and could therefore transmit untrue state updates to the server for his own benefit. The liaison function should therefore have the possibility of being rotated among several peers in the group, either through a round-robin or random assignment algorithm.

However, since liaisons utilize more network resources than other peers, due to a connection with the server, it might be beneficial to choose clients with high bandwidth to perform this function. The choice of the liaison assignment algorithm will therefore depend on the nature of the environment built on top of Virtutopia. Currently, liaison assignment is done only when a peer group is formed, and the first peer becomes the liaison. When this peer disconnects, the liaison is chosen based on seniority in the group.

Peer Connection

After a user is authenticated by the server, the user's client receives a message specifying the network addresses of its peers. In the case of the first client in any peer group, its address will be the only one in the message received from the server, and the client will become the liaison. When the next client joins the peer group, she will be sent a message with the addresses of all her new peers. At the same time the server will send all existing peers a message with an updated state of the peer group, including the newly connected user. Each of the existing clients will add the new user's network address to its list of peer addresses and await its connection. The new user will then connect to all its peers, and the peer group will become fully connected, with each client connected to all its peers.

Leaving a Peer Group

When a user disconnects from the server or exits the game, a quit message is sent to the server. The server updates the internal state of that user's peer group, removing the appropriate client, and then sends a message with the new state of the group to all remaining peers. The peers parse the message and notice that their group state contains a peer that is no longer connected, and remove the appropriate record. If the disconnecting client was the liaison, the server will automatically assign a new liaison for the group, based on the liaison selection algorithm. The new liaison upon the receipt of the message with updated peer group state will start performing the liaison function.

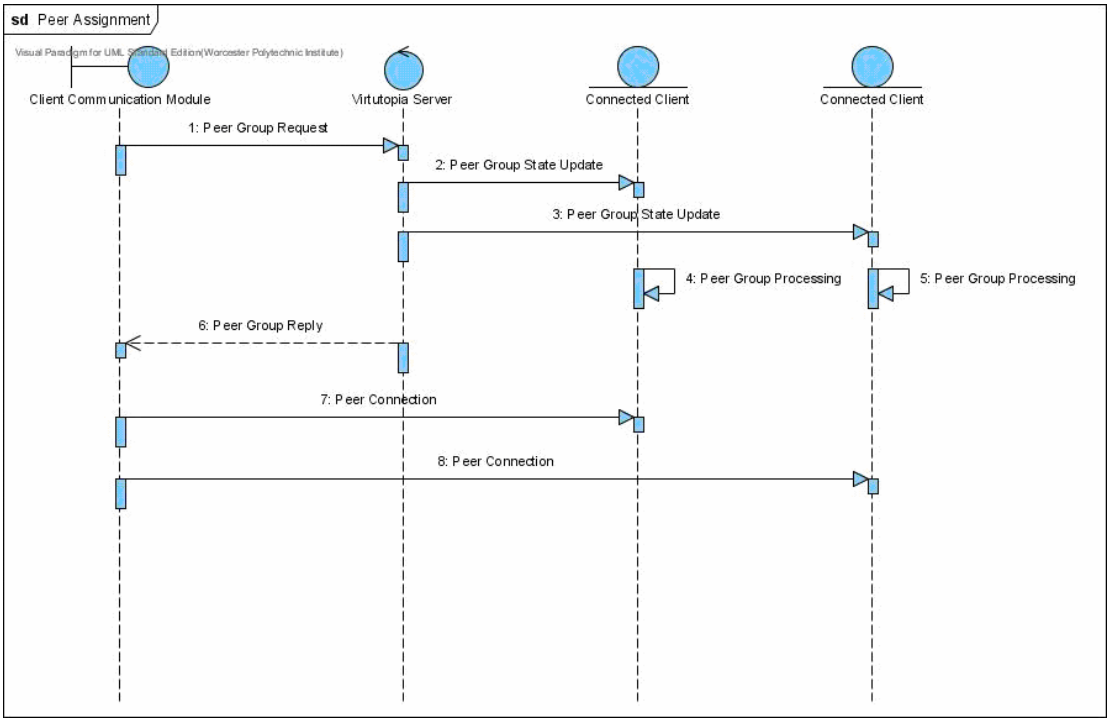

Figure 6.5.1-2 1 Virtutopia Peer Assignment

Parameters and tradeoffs

1. Maximum number of peers in a peer group. Currently: configurable.

A high number of peers in a peer group allows the server to spend less per-user resources, decreasing load on the server and increasing scalability. However, the bandwidth necessary for the communication between the peers grows as the square of the group size. It is therefore important to choose a small enough number so that all peers have enough bandwidth to communicate with their group.

2. Peer group assignment algorithm: configurable. Currently: based on connection order.

There are several different peer assignment algorithms: in-game location, network proximity, bandwidth available to clients, etc. The choice of algorithm will largely depend on the application.

3. Liaison assignment algorithm: configurable. Currently: based on connection order.

A liaison consumes slightly more resources, as it needs to forward game updates to and from the server. However, a liaison has more possibilities to abuse in-game rules since it is the only client in its group communicating with the server. Liaison-rotation schemes might be necessary for some applications.

6.5.2 Intra-Server Protocols

Because of the Virtutopia framework's reliance on QNX Message Passing to modularize the server application, several communication protocols had to be devised to facilitate effective communication between the various modules.

The protocol for requesting object type information from the OTR module requires that the initiator (the one making the request) supplies information indicating that the operation 'find object type' is desired, and also supplies the text string representation of the object type's unique identifier, i.e. world.object.ball$. The OTR module performs the requested operation, and then sends the initiator a text string that encodes the properties that the object type has. Along with this reply is an indicator of whether or not the operation succeeded, i.e. the object type exists, or it doesn't.

The protocol for interfacing with the Runtime State Module supports the following operations:

- STATE → Request the entire game state
- GET → Request a single specific object instance
- UPDATE → Submit a change to an existing object instance
- DELETE → Remove an object instance from the world
- CREATE → Insert a new object instance into the world

For the state operation, the initiator specifies the region of the world that he is currently in, and the RTS module provides a string-encoding of every object in the world zone that

the initiator specified. For the get operation, the initiator specifies the unique instance identifier of the object it would like to see, and the RTS module returns a description of that object instance if it exists, or an indicator that the operation failed if the object instance did not exist. For the update operation, the initiator specifies the unique instance identifier of the object instance that it would like to update, as well as the new state of the object, encoded into a text string. The RTS module replies with an indicator that tells whether or not the operation succeeded. For the delete operation, the initiator simply sends the unique instance identifier for the object it wants to delete, and the RTS module will reply with an indicator of whether or not the operation succeeded. For the create operation, the initiator sends the unique identifier of the object that is to be created, and the initial state of that instance, encoded as a text string. The RTS module will attempt the operation, and return an indicator of whether or not it succeeded.

The protocol for interfacing with the Authentication Module is simple. The initiator specifies the username and password pair for the desired login, and the Authentication Module looks for that pair in the database, and then replies with an indicator of whether or not the operation succeeded.

The protocol for interfacing with the Resource Module supports the following operations:

- GET → Get a resource, or resources
- ADD → Inform the module about new resources

For the get operation, the initiator specifies whether or not he wants the entire set of resources, or just a single resource file. If he wants a single file, he also supplies the filename of that file. The Resource Module then finds all of the files and sends them to the initiator (if they are text-based) or sends them a file system link so the initiator can load them itself (if they are binary). The Resource Module also sends an indicator that reports success or failure of the operation. For the add operation, the initiator specifies a filename, and the Resource Module adds the filename to the master list of resources if it is not already there. As a reply, the Resource Module sends a success or failure indicator.

6.5.3 Engine Communication Protocol

The engine communication protocol consists of a simple set of messages. The primary message type is the update. Updates contain the dynamic properties of an object being

changed in a given frame. Currently they include object id, position, velocity, rotation, and scaling, the latter two encoded in a transformation matrix. Add and remove messages are used to add and remove objects from the engine, such as when they are created or deleted within the client. Add messages include static properties of the object, currently the object id and the model name. Additionally, there are command messages, which consist of a command and an argument string. Currently supported commands are load, which will tell the engine to load a level file, and quit, which forces the engine to exit. A quit message is automatically sent whenever the Engine Communication module is told to disconnect from the engine.

The engine can also send messages back to the client. These messages are user input updates. A user input message consists of the keyboard button and its state. Such messages are created any time the user presses or releases a button. Each message sent in either direction includes a small footer string to ensure that each message is complete. This is necessary to prevent the processing of incomplete messages, because communication is handled asynchronously and it is likely that that last message in the buffer will be cut off before it has been completely transmitted.

6.5.4 Event Management Protocol

An event represents either a change to a single object in the runtime state or a creation of more events. Events are stored in an event object, which contains multiple data fields. The data field ‘description’ is the only required field for the event to be considered a valid event. The event description is what is used to determine how to handle each event and is therefore necessary to be included. Events only handle two objects, so in order to handle an interaction between more than two objects, more than one event must be used. Events that are intended as direct changes to the runtime state must state one or two objects to be modified and the position, velocity, and acceleration values for each of the objects that are to be changed. Events that are intended to trigger a script by the Behaviors module must contain all of the information that the script requires, excluding the object information. An example of this would be an input event, which requires that

the event contain the keyboard key that was pressed in order to properly evaluate the event.

6.5.5 Lua Script Interface Protocol

Because of the limitations of Lua interfacing with C++, there are certain protocols to writing a proper Lua script for the Virtutopia project. Remember that all scripts are run in response to events. Because of this, a script name can be generated from the event that you wish it to be run in response to; specifically `onEvent.lua`. That is, a script to be executed in response to a Collision event is named `onCollision.lua`.; a script that responds to Move events would be named `onMove.lua`. A complete list of events used in the Virtutopia system can be found in Appendix A.

The script must contain a "main" function that is to be executed by the core code. The core code will pass it the two objects contained in the event that the script is responding to. The script can then call back into the core code to retrieve any information related to the objects that it may need, using a function that the core code registers with the lua stack.

When the script is complete, it returns a set of data for the core code to read in and write into an event. This event is then sent on to the Event Manager for later evaluation.

6.5.6 Object Transmission Protocol

Object Instances are transmitted often over the network many times; for example when the server provides the current game state to the client machine or when a peer sends out a game state update to another peer. The transport method must be reliable and it must also easy to debug. The method that we developed was to pack each object, along with its properties, into a text string that would be sent over the network to the receiving side.

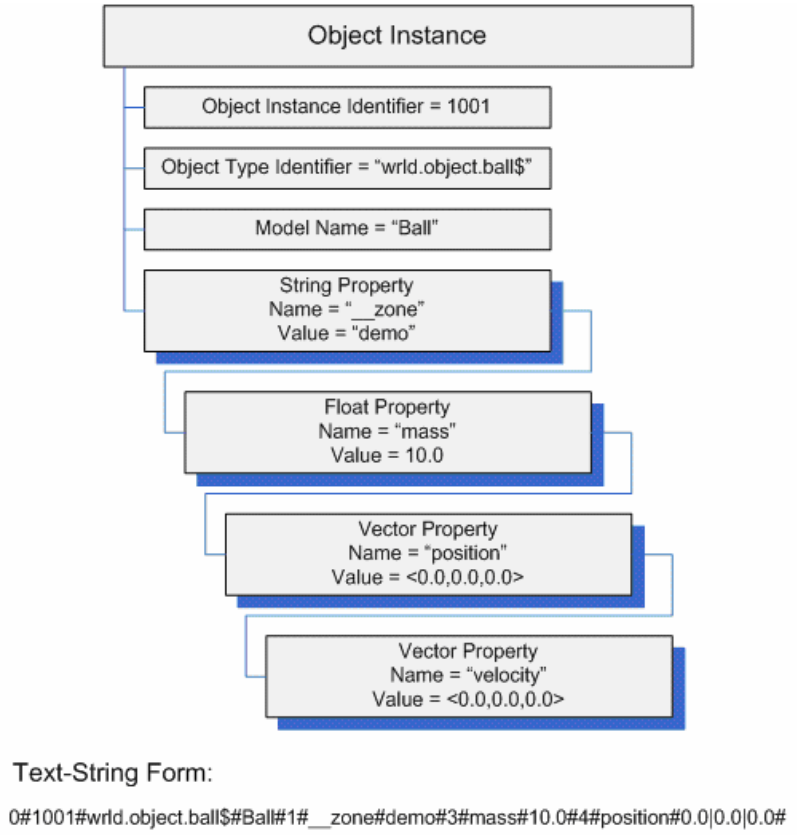

Figure 6.5.6-1: Object Instance Architecture with String Encoding

This method worked flawlessly, and in addition, it was easy to debug during development. Because the text string could be trivially printed out, or displayed on the screen, we could view its contents to determine if it contained the proper data. This method, however, has serious performance drawbacks. The string that the sender sends out has to be assembled before it can be sent, which introduces additional overhead. Also, the receiver must parse this string, which also introduces overhead. Finally, a significant amount of network bandwidth is wasted because the string can vary in length from update to update, depending on the contents of the object to be sent. For instance, the string that represents a floating-point value is of arbitrary length (the length of "3.07341" versus the length of "3.2") despite the fact that the floating-point value only consumes 4 bytes of memory on the host machine. The solution for this inefficiency would be to save bandwidth and processing time by using a more efficient algorithm for using a binary format when transmitting our object instances over the network.

7. Testing and Evaluation

The utility of any computing system is dependent on the performance results of that system in real-life usage scenarios. To evaluate the behavior of the Virtutopia client and server, we designed and executed several performance tests. Each test was created to measure a particular aspect of the Virtutopia infrastructure.

7.1 Maximum Number of Connections Test

7.1.1 Description

The purpose of this test was to determine the maximum number of simultaneous active connections to the server which can be sustained. To achieve this, a Linux client program and shell wrapper were designed. The shell program received as input the number of client processes to create; each client process connected to the server and then maintained an idle connection. To execute the test, clients were added continually until the server wasn't able to receive new connections.

The maximum number of active connections was expected to be slightly below 1000, which is the maximum number of open file descriptors per user allowed by the default configuration of the QNX server. Sockets are described using file descriptors in Unix systems, so the number of open files and sockets can not exceed 1000. Since the server process opened file descriptors for log files and standard input and output, as well a separate socket for a thread listening for network connections, the number of open connections could only be strictly less than the maximum number of per-user sockets.

7.1.2 Results and Analysis

The test was executed by launching command-line clients connecting to the Virtutopia server from multiple Linux hosts on the WPI network. Each machine launched between 50 to 100 client processes within a short interval, as fast as new processes could be created.

The test was executed three times, and the number of simultaneously connected clients was found to be 291, 251 and 252. In each case, after processing the final client the server stopped responding to new requests. However, the server machine was still fully responsive and allowed the termination of all server-side Virtutopia modules. The cause of the 'locking' behavior, as well as the discrepancy between the ideal maximum number of connected clients and the observed values have not been fully determined. It is speculated that another operating system limit disallows the allocation of new resources to the server process.

Despite the fact that the server did not reach our expected number of connections, 250 active connections is still an impressive result because it represents the number of peer groups that the server can sustain. Thus the server can theoretically sustain a number of clients equal to 250 multiplied by the maximum peer group size.

7.2 Maximum Peer Group Size Test

7.2.1 Description

The purpose of this test was to determine the maximum number of peers in a single group. To achieve this, the Virtutopia Windows client that did not output to a graphics engine was used. This client logged the amount of data that it sent and received, as well as its current peer group size. All clients connecting to the server were placed in the same peer group.

In order for a specific number of peers to be determined a success, the upload bandwidth usage must be lower than that of an average cable modem, which is defined to be 384 Kbit/s. The test was executed by connecting one client every ten seconds until a total of thirty-two clients were reached, where each client logged all network data to its own log file. The expected maximum number of peers in a single group is nine, selected using a formula that represents a peer group's optimal bandwidth usage:

B: Number of bytes the liaison can upload per second

S: Size of the update to be sent

T: Number of times per second the update is sent

In our test, we sent approximately 200 byte updates 30 times per second with the liaison sending a maximum of 48,000 bytes in a second.

N: Maximum number of peers in a single group.

$$(\mathbf{B}) / [\mathbf{S} * \mathbf{T}] = \mathbf{N} - 1$$

This equation resolves to N = 9 using the given values.

7.2.2 Results and Analysis

The results of the test are plotted on the graph below.

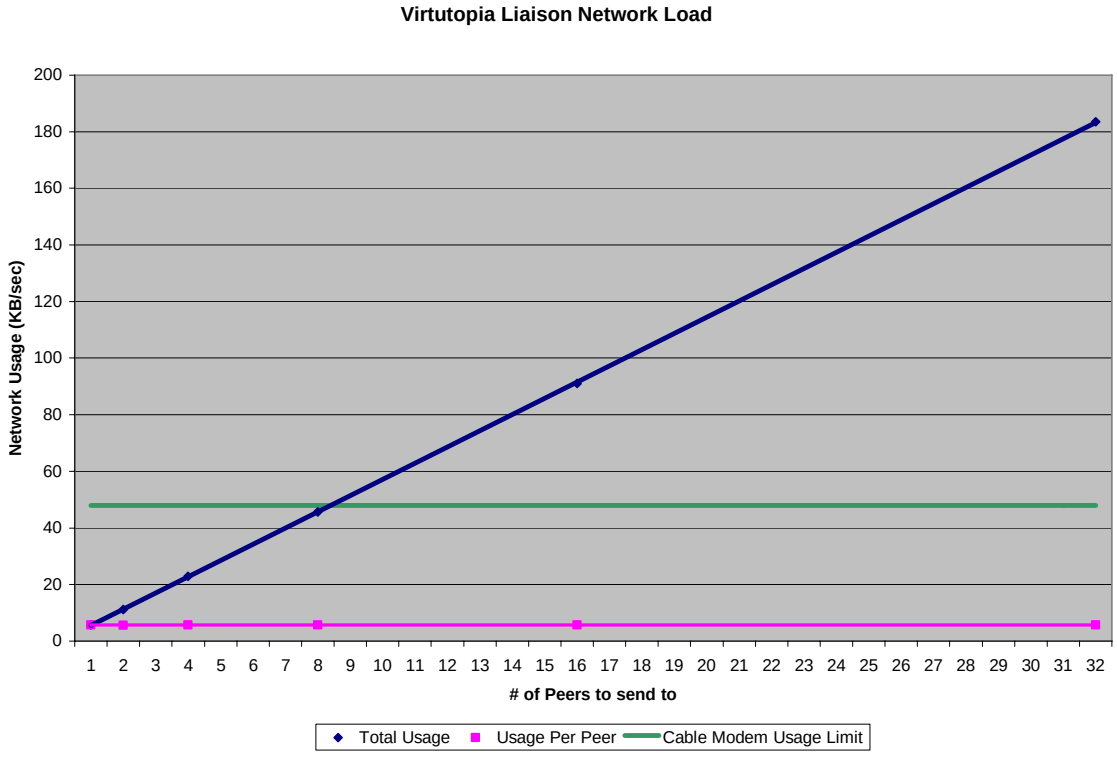

Figure 7.2-1: Virtutopia Liaison Network Load vs. Peer Group Size

Bandwidth usage reached the specified maximum cable modem usage limit when the number of peers to which data was sent exceeded eight. Thus the resulting peer group size, including the liaison, is nine peers, which is approximately the expected result. The reason the data did not exactly match the expected result was likely because the size of the updates involved some slight variation, ranging between 195 and 197 bytes. Drawing

from this result and the result from Section 7.1, the maximum number of clients connected to the system is 2,250. This number is derived from the fact that the total number of users is the number of peer groups multiplied by the number of peers per group.

7.3 Maximum Number of Objects Test

7.3.1 Description

The purpose of this stress test was to see how many moving objects the Virtutopia client is capable of processing at an acceptable frame rate. To achieve this, several test realms were created, containing various numbers of balls in an otherwise empty room. The faceless and C4 Virtutopia client applications were then executed. Each client connected to the Virtutopia server, downloaded the runtime state, which contained 6 walls, and 1, 4, 9, 16, 25, 36 or 100 balls, and then ran each simulation for one minute. The Update Manager module logged its frame rate over time to a text file, so that we could analyze the results of the tests.

Our expectations for this test were not high. It was known that the physics module was not optimized for large numbers of objects because there is no spatial partitioning mechanism to accelerate the physics calculations. Our collision detection algorithm therefore runs in quadratic time, severely impacting performance once enough objects are processed by the client.

7.3.2 Results and Analysis

The results of this stress test are plotted in the graph below.

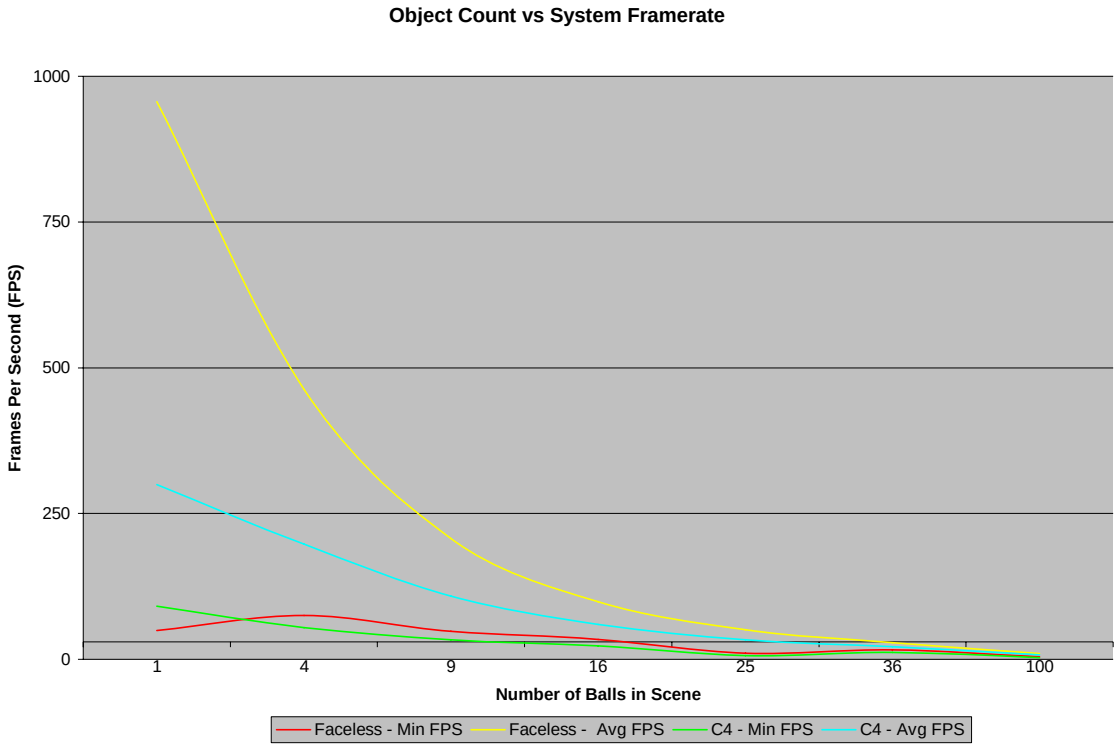

Figure 7.3-1: Client System Framerate vs. Objects Count

Results were gathered and plotted for both the faceless client, and the full client including the C4 engine showing the state of the virtual environment. For both cases, both the average and the minimum frame rates are shown. Our primary performance measure is the average frame rate, as we are willing to accept temporary drops in performance, as long as the overall behavior of the system is satisfactory.

As was expected, the average performance of the faceless client is significantly better than the graphical client, due to overhead incurred by the C4 engine process. However, as the number of processed objects increases, the frame rates converge, and for 36 objects, the frame rates are almost equal. Taking into account our goal of maintaining a frame rate of at least 30 frames per second (so that the human eye perceives the rendered world as 'smooth'), the number of objects which are processed by the client must be smaller than 25.

This result confirms our expectation that the lack of a space partitioning scheme in the physics module will cause a quadratic reduction in the frame rate. The implications that

this result has on the Virtutopia project is minimal. While a mere 25 objects is an exceptionally small quantity, this does not discredit the concept of Virtutopia. Essentially, our current implementation of the Physics module has no form of space partitioning, and therefore is very inefficient, because it runs in quadratic time. Optimizing Physics to utilize an acceleration structure, such as a Kd-tree or Bounding Volume hierarchies, is outside the scope of this project (See Sections 8.4 & 8.7). This method is used in modern video games and graphics engines to increase performance. Implementing such an acceleration structure would increase performance of the system, and allow for greater scalability with respect to the number of objects.

7.4 Maximum Server Load Test

7.4.1 Description

This test was designed to measure the network load on the server under the worst-case condition of all peer groups occupying the same in-game area. In this scenario the server acts as a repeater between all groups' liaisons to enable clients to periodically synchronize their state information with the server and all other clients, eliminating inconsistencies introduced due to peers only communicating within their own peer groups.

During each network update cycle (defined for the purpose of this test to be 1/30 of a second, to be consistent with the client event manager time slice) the server receives a typical 196-byte state update from each peer group and merges all updates into one message, simulated for this load test by picking only one group's update each cycle. This message is then sent to all groups' liaisons.

The 196-byte message size was chosen to reflect a typical object update size in the current Virtutopia system, and also the maximum message which can be sent 30 times per second to all peers in the optimal peer group of size 8. While larger messages might be possible, typical client bandwidth limitations will create the need for more sophisticated update-sending and merging schemes.

Four tests were executed, with the number of co-locate peer groups ranging from 20 to 125; in each test the peer group size was set to 1, so that each client performed the liaison function and was sent updates by the server. We measured the number of messages processed by the server each second and the server bandwidth necessary for the server to send and receive updates from each peer group. The tests were performed with the assumption that a typical Virtutopia server will be connected to the Internet with a T1 connection, allowing for a maximum bandwidth of 1.544 Mbit/s.

7.4.2 Results and Analysis

The results of the measurements are shown in Figures 7.4-1 and 7.4-2.

We found that the number of messages processed and the bandwidth utilization rose linearly, indicating that all messages were sent and received properly, until a saturation point was reached, at which the server could not keep up with the sending of updates to all groups. The processing limit for the server happened at about 62 peer groups (processing 62 * 30 ~= 1800 updates per second, or about 2 updates/millisecond).

An interesting observation is that for a larger number of co-located peer groups, the number of messages which can be processed by the server in each update cycle decreases. This seems to be caused by the fact that the server must do extra processing for each message, iterating through the list of all connected peer groups and sending data to them, which causes processing overhead.

The result shows that using commodity PC hardware such as the current Virtutopia server, it was possible process more messages than the number allowed by the bandwidth of a T1 connection. The T1 bandwidth allows for processing of about 35 peer groups located in the same area, under the stated assumptions that each update occurs 30 times per seconds and is contained in a message of size 196 bytes.

It is predicted that with a larger message size, the number of messages processed by the server would remain close to the number of small messages which can be processed in each update cycle, as most overhead was found to be present in iterating through all peer

groups and forwarding the data to them. Therefore servers with faster network connections (such as 10Mbit/s) are predicted to be limited by the CPU rather than bandwidth.

For this test, the server simulated a mechanism for merging the updates from all peer groups into one meta-update, which demonstrates the utility of such a mechanism in Virtutopia. If each group's update was forwarded to each other group, the bandwidth and processor utilization would grow quadratically, limiting the number of co-located peer groups to less than 8, and significantly system performance for this case.

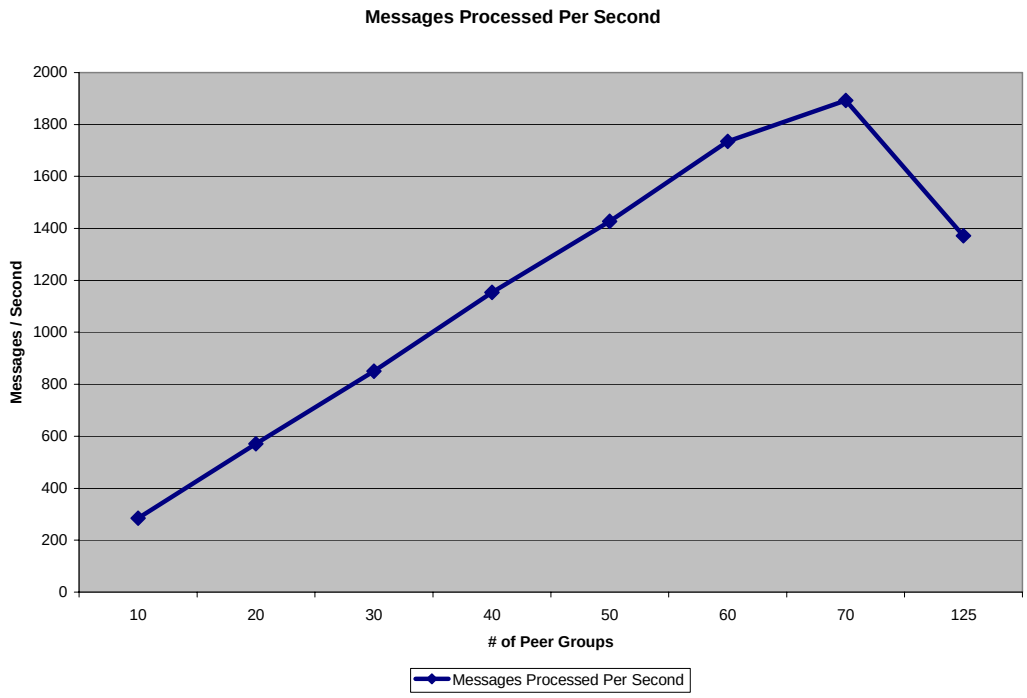

Figure 7.4-1: Server-processed messages per second

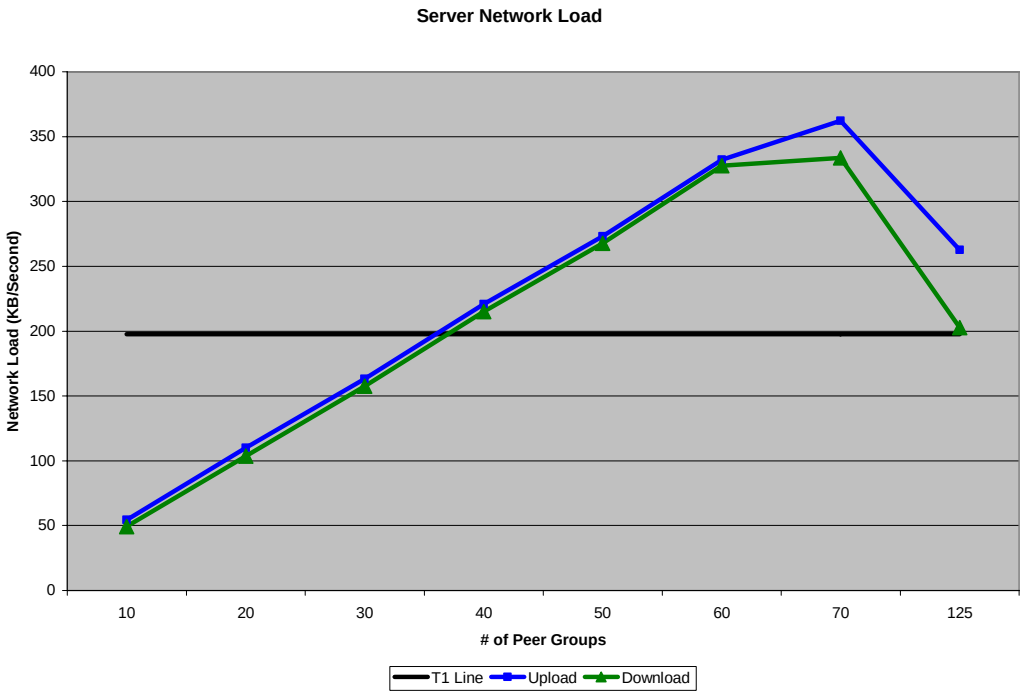

Figure 7.4-2: Server bandwidth

7.5 Distribution of Client Processing

7.5.1 Description

The purpose of this test was to determine the distribution of work that takes place in the processing of a time slice. To achieve this, we used a microsecond accurate timer and recorded how long each of the five steps to complete a time slice took to finish in addition to the total amount of time to process the time slice. We ran the test using both the faceless client as well as the C4 client, using three test scenarios for each with varying number of balls. We ran the test for a minute each time recording the results to a log file.

Our expectation for this test was that step 1, update, which contained all of the physics calculations, would use the largest percentage of the time slice. We expected this to be especially true as the number of objects increased.

7.5.2 Results and Analysis

The results of this test are plotted below.

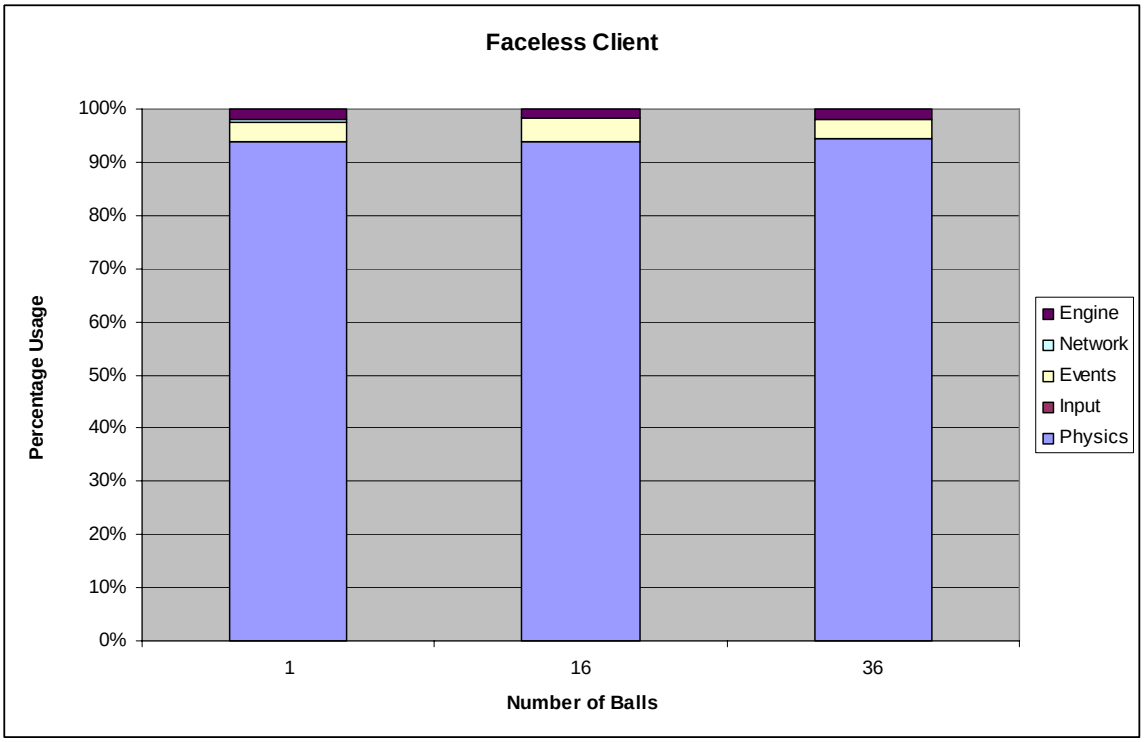

Figure 7.5.2-1: Client Workload Breakdown vs. Number of Balls (faceless)

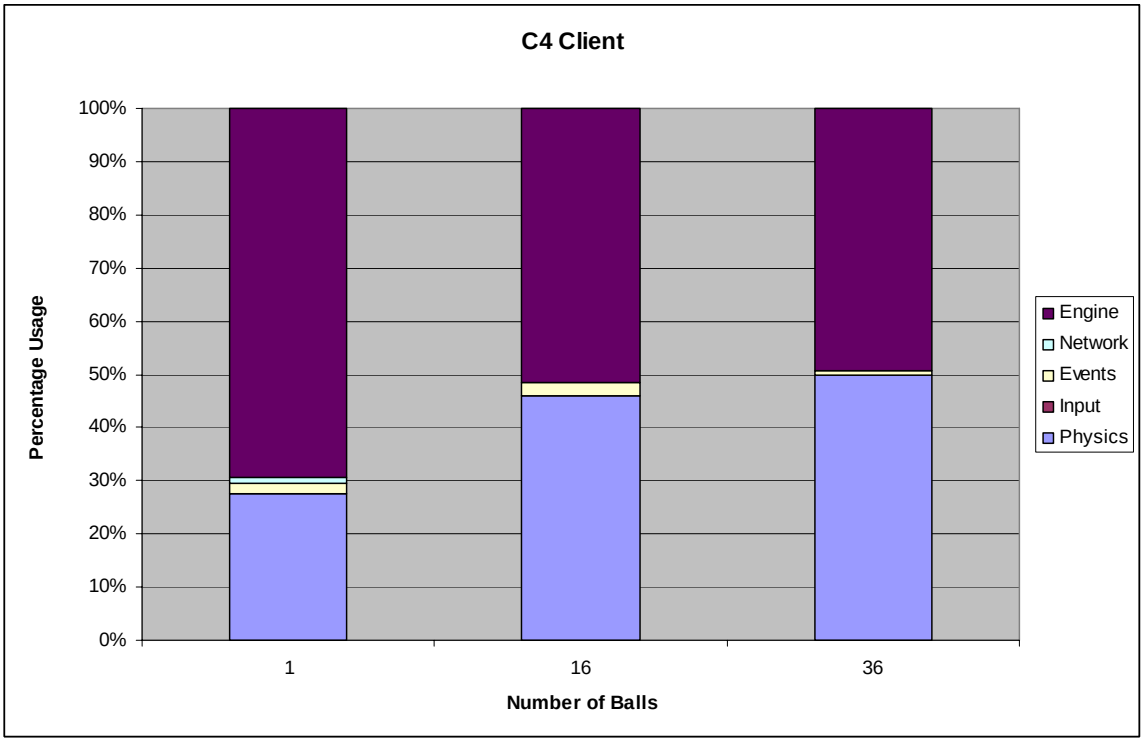

Figure 7.5.2-2: Client Workload Breakdown vs. Number of Balls (C4)

The tests with the faceless client ran as we expected. Physics uses the vast majority of the time to create a time slice with processing events coming in at a far second. However, the C4 client had some surprising results. The updating of the engine uses at least as much processing power as physics. We attribute this to either HawkNL not being efficient or socket code in general not being terribly efficient.

This brings to light a new optimization problem because sending the object information should not be as costly as the tests show. This is definitely a problem that should be reviewed in further iterations of Virtutopia to improve performance.

8. Future Work

8.1 Authentication Module

The Authentication Module on the server could be enhanced. Currently, the Authentication Module accepts any login request that comes in, as long as it has a valid username and password. This system was sufficient for our testing and development needs, but in the future, it should maintain a list of users that are currently logged into the system, and only allow logins from accounts that are currently not logged in.

8.2 Communication Module

The current state of the Communication Module provides Virtutopia clients with the ability to reliably communicate with the server and within their peer group. Future additions to this module, both on the client and server side, will concentrate on developing more elaborate algorithms for network-based interaction.

Currently the server-side Communication Module assigns users to peer groups whenever they connect to the server, before any game messages are exchanged. Some environments might benefit from assigning users to peer groups based on information supplied by the client in the course of the connection-handshake process (such as the current level or game-world region), or retrieved from the server database (last location of user in the world before disconnecting). Therefore new algorithms for choosing a peer group for a client will be beneficial, increasing the flexibility of the system.

Adding a more robust peer group assignment policy enabling the server to group users by game region will also allow peers to connect to different game regions using the same server. The server could then easily contain separate and independent areas within the world, allowing several peer groups to perform the same game-related tasks or missions in parallel, without affecting other groups.

Another algorithm which could be extended is the one responsible for choosing a liaison client for a given peer group. The current algorithm chooses the first client to join a peer group to become its liaison, and this client remains the group's liaison until it disconnects. In order to help prevent cheating attempts, the liaison function could be rotated among peers in the group, either randomly, or using a pre-defined order. An important consideration is that the liaison will utilize more CPU and network resources than other peers; clients running on slower hardware might be barred from acting as a liaison, whereas faster clients could have a higher chance of becoming the liaison.

The client Communication Module could be improved by adding a queue for outgoing messages. Currently the Communication Module is delivering network messages to the Update Manager by putting them in a thread-safe queue; however, handling of outgoing messages is done by the Update Manager explicitly calling a Communication Module function to send the data. Instead of a direct function call, this process could be done by putting each message in a separate outgoing queue. In addition to increasing the separation between the two interacting modules, it would allow the Communication Module to contain logic determining the appropriate parameters for sending data, such as the interval between updates and the number of messages to send each time. The logic could be based on each message's recipients, customizing the frequency of updates sent to the server and each peer.

The initial design of the Communication Module also included two sub-modules performing data encryption and compression tasks. Although placeholders for appropriate algorithms are in place, those sub-modules have not been implemented and therefore the Virtutopia network protocol does not support encryption or compression. Future work on implementing these functionalities should also focus on their effect on system performance.

8.3 Object Type Repository

In its current state, the Object Type Repository client side module is feature complete. On the server side, there are a small number of changes that would improve the performance

of the OTR when large numbers of clients are connected to the system, and when there are a large number of Object Types in the system. The current server side module keeps all Object Types that are ever requested loaded into system memory as long as the module is still running. This is sufficient for a small number of Object Types that exist in a given world, but for larger types, the server machine could run out of working memory as more and more Object Types get loaded into the system. A solution to this problem would be to have a tracker on each Object Type that kept track of how long it had been since that object had been requested by another module. When a predetermined period of time expires, the Object Type is unloaded from system memory. If that object is ever needed again, then it can simply be reloaded into system memory. This way, if objects are needed often, they stay in main memory and are readily accessible. If, however, an Object Type is rarely needed, it can still be accessed from disk, but it won't take up space in system memory for long. The implementation for this algorithm would likely come from the field of Operating Systems, such as a paging algorithm.

Another useful enhancement to the Object Type Repository would be to upgrade our XML parser to support DTD Doctype Definitions. This way, the system could automatically detect any syntactical error in the XML for a given Object Type. During the implementation, we ran into a few bugs that were caused by invalid XML files being used for Object Types. A single typing mistake or other error could make the system improperly load Object Types from disk. When the errors in the XML files were resolved, the bugs went away. A simple upgrade such as a validating XML parser could make detection of such XML errors much easier. This would allow us to track similar bugs more quickly in the future. Also, a validating parser could be used as part of an Object Creator or World Creator application that developers use to create and populate their virtual worlds. If the validating parser were used here, the application could act as a "compiler" that would check the developer's XML files for errors.

8.4 Runtime State Module

The Runtime State Module could also benefit from some enhancements. On the client side, the only time object instances are removed from the Runtime State is in between the

segments of one time slice. That is to say, object instances are removed after processing physics, or after processing events, or after sending off network updates. The reason for this is that removing an object from the list would invalidate the iterator objects that are used for traversing the list of objects, causing a segmentation fault. The solution to this is quite simple in theory, although it is currently unimplemented in the system. This solution is to mark removed objects in the list as ‘removed’, but not actually free their memory and delete them from the list. Instead, at the end of every frame, a garbage collector can sweep through the list and remove objects that have been marked as ‘removed’. This feature would allow the system to remove object instances at any arbitrary time without invalidating any iterator that some module is using to traverse the object list. The only additional complexity in traversing the list of objects is that each operator would be required to check this flag before performing its work on the object instance.

Another useful enhancement would be to have the object instances in the list sorted by their location in the game world. If the object instances can be sorted using some form of space partitioning algorithm, such as a KD-tree, or bounding volume hierarchies, then the system can split the object instance list into several mutually exclusive sets. This, combined with the marked-for-deletion system described above would allow the update manager to be multithreaded without requiring synchronization objects to control thread access to object instances. One major advantage of multithreading without the need for synchronization objects is that the system can scale to take advantage of multiprocessor systems, while avoiding the overhead involved with those synchronization objects. The downside to this, however, is that the reconstruction of the space partitioning could be expensive if it needed to be done every frame.

The server-side Runtime State Module could also benefit from a caching feature where the current state of the world is written to disk periodically. This would allow the RTS module to suffer a system crash, and the state of the RTS would be preserved, with only a small amount of time’s worth of changes lost.

8.5 Update Manager

The Update Manager has several areas that can be expanded, most of them being related to performance. As it currently stands, the Update Manager is a single thread which contains a lot of processor-intensive operations. While this is fine for a single-core processor, dual-core processors are becoming prevalent and therefore it is a good idea to optimize the system's code to utilize them. Reviewing the five step process of the Update Manager, although we can see that the order of steps in the time slice must be maintained, the steps themselves have a lot of possibility for parallel operations. For example, the third step 'Process network updates' is purely parallel. This step is where the Update Manager reads each node from the queue of updates and writes them to the runtime state. Each individual update has no specific relation to any other, so creating a thread per core would allow the operation to be completed much faster than before. This multi-threading could also be applied to the other four steps, however steps one and three have complicated interactions that can occur while running parallel, which therefore would merit a significant design around allowing this increase in performance while assuring the system works as intended.

Another area to improve would be how it is determined that an update must be sent. Currently, after a set time interval, the Update Manager sends a network update for every object in the Runtime State, regardless of whether or not its state has changed. A better way of implementing this would be perhaps marking when an object has changed and only sending the network update if it is marked as changed. It should be noted that the time interval should still be implemented as a maximum number of network updates to be sent per second or else the client may run into bandwidth issues given a standard cable modem connection.

One more improvement would be changing how updates for objects are stored before being sent over the network. Currently, the entire object information is being sent in each update, even if only a few properties have changed. To reduce the amount of bandwidth used, the network updates could instead only contain the changes made to the object.

8.6 Event Manager

The Event Manager has a few areas that can be improved upon. The first improvement would be allowing events to have dynamic attributes instead of specific ones. An example of this would be replacing all of the class variables with a linked list of attributes. This allows events to have only as much data as they need, but also allows them to add attributes without having to bloat the event class.

Another improvement would be making the event queue thread safe, which could then allow the Update Manager to multi-thread both the creation and the processing of events.

8.7 Physics

The Physics module contains a few areas for improvement. It uses a 'moved' variable to determine whether or not it has already acted on an object, as a quick reject. Removing this and replacing it with an algorithm for only acting on each object exactly once would be a good improvement. It could also be improved by various optimizations, specifically to the collision detection algorithms – by its nature, collision detection must scale exponentially. By breaking it up such that each object only checks against the objects it is capable of colliding with, the 'n' is reduced and the system scaling is much improved.

8.8 Behaviors

The Behaviors module supports all events having a script associated with them, to be run to determine the outcome of the event. At the current time, only a few events have associated scripts; this will need to be expanded to cover all events. The Collider class family, a part of Behaviors, also contains the actual collision detection algorithms for each object type, which could be further optimized. In addition, a function for Lua to create objects is currently a stub, and it may be ideal to expose other functionalities to Lua by registering functions with the scripts.

8.9 Engine Interface

The engine interface, as implemented in this project, exists primarily to allow for the testing of other modules. The top priority was to simply make it functional. As a result,

little attention was given to the issues of managing content or providing a robust user interface. There are many important ways in which future projects should build upon the existing framework in order to produce a robust user experience.

One important topic that has been left unexplored is the importing of content. In order for Virtutopia to maintain its engine independence, it will be necessary to create methods for importing game assets into the running engine. This is a complicated issue, because different engines will have different requirements for models, textures, and sounds. We also wish to allow developers to use any format they choose when creating Virtutopia realms. We have not reached a good general solution to this problem in our design.

While the module interface is fairly simple and robust, the communication between client and engine is underdeveloped. Currently, little information is actually sent to the engine. This is mostly because we have little information to send in our present project. However, as the client evolves and new behaviors are made possible, it will be necessary to extend the types of information sent. The best way to achieve this would be to completely serialize the object types and instances when sending them to the engine.

This would allow different engines to make use of the game objects in different ways, such as by adding special effects for objects that carry certain properties. Due to the highly dynamic nature of the objects in the client, it would then be easy to add new properties at will. There are also performance considerations to be taken into account with this approach.

Another essential feature is a flexible user interface. While simple interaction with keyboard and mouse is possible under the current implementation, it will be necessary to support other types of input in order to produce compelling virtual experiences. It is difficult to do this in a general way. Also, the user interface must extend outside of the virtual world, so that the user may have control of the application. The game engine should have some access to the command structure of the client, so that the engine may

implement a GUI for using the system, such as for connecting to servers or managing user data.

Lastly there is the C4 engine module itself. This was not given much attention, since it was only required to perform basic tasks for the purposes of this project. Aside from the client communication code, which has been designed such that it can be used in any engine written in C++, most of this code should not be used in the future. The implementation is highly specific to this project, and would not be easy to extend. Future projects will wish to have engine "glue code" that is more reusable, and our implementation has a few severe limitations. In the case of C4, this will require more drastic changes to the architecture of the engine than have been made here. C4 is designed to be easily extended, but completely replacing parts of it can be somewhat complicated, and will require more time.

The engine interface is in some ways the most ephemeral of all the modules. It is specified to perform complicated input and output tasks, but has no knowledge of how the game world behaves. These two aspects of its design are often in conflict, and a balance is difficult to find. As a result, it is unclear how it will evolve as Virtutopia continues to be developed.

9. Conclusions

Virtutopia began with a lofty set of goals. Our intention was to produce a reasonably complete framework, upon which future projects could build in the years to come. The result is that we did not focus on building any particular module to completion. Instead, we had to create a working skeleton for each module. These skeletons had to be able to communicate with each other in a manner that is correct with regard to the overall design, so that the system will function correctly as the individual modules continue to be developed in future projects. Therefore, the main challenge of this project was to design a strong application programming interface (API), which is the set of classes and interfaces which future programmers will use to add functionality to the system.

In this task we have mostly succeeded, although there are instances where we have had to short-circuit our design in order to produce a working implementation within the time constraints of the project. This is again due to the ambitious nature of Virtutopia. To produce a framework of this scale, even with minimal functionality requires a great deal of integration work, and we have had some difficulty apportioning our time toward the end of implementation. This time pressure has lead to some questionable implementation decisions, but we have also tried to update the design whenever necessary to incorporate the lessons we have learned from these problems.

As for the implementation itself, we have had mixed success. Our intention was to use iterative test driven development. This practice was used in a somewhat inconsistent manner, but was generally useful. Having development milestones helped us keep the project on track, and unit tests made it easier to verify the integrity of our code and to measure progress. In particular, the test framework made it simple to run and debug different branches of the code without having to add new entry points into the program. The main problem we had with test driven development was in keeping old tests functional. Toward the end of production we were deprecating large portions of the code base. Particularly with the runtime state, much of which was moved to the server from

the client, some tests were rendered irrelevant or unusable. This resulted from the need to have working demos before certain vital functionality was in place.

Our testing procedure was created for a single purpose: to evaluate the hybrid architecture upon which our project was built. The theory of this architecture is sound, but producing real world results is difficult with an incomplete implementation, and without user studies. To account for this we devised tests that would stress the client and server under both ideal and worst-case scenarios. To achieve this we had to simulate the traffic to some degree. Our implementation does not include key features such as object ownership, which has a large effect on where data is being sent and to whom. We also use a naïve approach to sending state updates between peers, where the entire state is sent at regular intervals. However, due to the sub-optimality of our approach, we are able to produce test results that are valid in the worst case and serve as a baseline for future improvements.

Subsequent projects will be able to use our design and our test results to implement a more robust system for distributed virtual environments. The hybrid architecture has proven to allow numbers of users that have been previously impossible using pure client-server or peer-to-peer networking, potentially opening the door for independent developers to build massive multiplayer environments on inexpensive hardware and with low-bandwidth hosting.

10. References

1. Chen, Alvin Yun-Wen. “A Hybrid Architecture for Massively Scaled Distributed Virtual Environments”. UCLA Multimedia Systems Laboratory.
07 Sep 2006. http://www.cs.ucla.edu/~alchemy/pubs/prospectus.pdf

2. Armitage, Grenville; Claypool, Mark; Branch, Philip. “Networking and Online Games: Understanding and Engineering Multiplayer Internet Games”. John Wiley and Sons Ltd, 2006.

3. World of Warcraft European Realm Forums.
http://forums.wow-europe.com/child-forum.html?forumId=11101

4. QNX Neutrino Operating System. www.qnx.com.

5. C4 Game Engine. http://www.terathon.com/c4engine/index.php.

6. Official Everquest Website. http://eqplayers.station.sony.com/index.vm

7. World of Warcraft Community Site. http://www.worldofwarcraft.com/index.xml

8. Blizzard Entertainment – StarCraft. http://www.blizzard.com/starcraft

9. The Programming Language Lua. http://www.lua.org

Appendix A: Types of Events

The following events are used in the Virtutopia architecture.

Move: A movement event updates the position of an object based on its velocity and acceleration.

Collision: A collision event performs the calculations for the new position and velocity of an object after a collision with another object.

WallCollision: A wall collision event performs the calculations for the new position and velocity of an object after a collision with a wall.

Input: An input event indicates that the user has performed some kind of input, and reacts accordingly.

Appendix B: Glossary of Terms

Authentication – Module that is responsible for ensuring that each player using the client to interact with the virtual world is a valid user of the system.

Behaviors –Module that governs creating responses for events by calling scripts.

Client – Application a user uses to connect to the server and run Virtutopia.

Communications – Module responsible for providing an abstraction for higher-level client and server modules, allowing each client to communicate with other players in the game, as well as the central server.

Engine – Application responsible for reading in input (such as keyboard) and producing output (such as graphics and sound).

Engine Interface – Module that connects the Update Manager to the Engine, allowing both input (such as keyboard) and output (such as graphics and sound).

Event - Either a change to one or two objects in the runtime state or a creation of more events.

Event Manager – Module that manages storing and accessing of events.

Faceless – Client that does not use a graphics engine.

Liaison – Special peer that acts as a gateway between the server and the rest of the peers in the peer group.

Lua – Scripting language that is used by the Behaviors module.

Mutex – Mutual exclusion object that allows multiple threads to synchronize access to a shared resource.

Object – Representation of a physical entity that exists in the virtual world.

Object Instance – Specific instance of an object.

Object Repository - Monolithic storage facility for every possible type of object in the world.

Object Type – Base type that is used to describe an object.

Peer – A single client.

Peer Group – A small group of peers connected to the server by a liaison.

Physics – Module responsible for processing Newtonian physics interactions such as movement and collisions.

Realm – Collection of virtual environments.

Region – Represents the client's interactive area.

Region State – Data that represents the client's representation of the world.

Resource Module – Module that is responsible for forwarding game or simulation resources to the Communications module so that they can be forwarded to the clients who request said resources.

Runtime State – Module that manages the world and region states.

Script – A program that is interpreted at runtime instead of compiled beforehand.

Server – Application that does the connecting, routing, and initializing of client's so that they can interact with the virtual world.

System – Module that governs initializing, starting, and stopping the client.

Thread - A portion of a program that can run in parallel with other portions of the program while sharing memory.

Time Slice - A time slice is a discrete period of time with variable length that is specifically started and ended by the Update Manager.

Update Manager - That starts, processes, and completes every time slice of the Client.

Virtutopia – An application and protocol for distributed virtual reality.

World State – Data that represents the server's representation of the world.