A RESTFUL ARCHITECTURE FOR MULTIUSER VIRTUAL ENVIRONMENTS AND SIMULATIONS

A Thesis
Submitted to the Graduate Faculty
of the
North Dakota State University
of Agriculture and Applied Science

By

Matti Kariluoma

In Partial Fulfillment of the Requirements
for the Degree of
MASTER OF SCIENCE

Major Department:
Computer Science

March 2014

Fargo, North Dakota

North Dakota State University
Graduate School

Title

A RESTful Architecture for Multiuser
Virtual Environments and Simulations

By

Matti Kariluoma

The Supervisory Committee certifies that this ***disquisition*** complies with North Dakota State University's regulations and meets the accepted standards for the degree of

MASTER OF SCIENCE

SUPERVISORY COMMITTEE:

Dr. Brian Slator
Chair

Dr. Simone Ludwig

Dr. Joel Ransom

Approved:

4/10/14
Date

Dr. Kenneth Magel
Department Chair

ABSTRACT

JavaMOO is an architecture for creating multiuser virtual environments using the MUD (Multi-User Dungeon) and MOO (MUD Object Oriented) design patterns (rooms and objects in the rooms, including "exit" objects that lead to other rooms). The MOO design pattern traces it roots back to the first multiuser virtual environments in the early 1980s.

The focus of this thesis is joining the MOO design pattern with a distributed architecture. A distributed architecture is pursued to reduce the per-server computational load, compared to a traditional single-server approach. The effect of transferring computational load to the client software is also investigated, with particular attention to the case of a graphical client with rich 3D visualizations.

This results in an architecture employing a RESTful (REpresentational State Transfer) common interface and non-authoritative state synchronization that supports the MOO design pattern, uses fewer server-side resources, and is deployable as a network of distributed servers.

ACKNOWLEDGMENTS

Thanks to Stephanie Sculthorp-Skrei, Dr. Brian Slator, Dr. Simone Ludwig, Dr. Joel Ransom, Otto Borchert, Robert Foertsch, Guy Hokanson, Adam Jacobs, Christopher Schaefer, Brad Vender, and Peng Yan.

Support for this work came from the Center for Science and Mathematics Education (Dr. Donald P. Schwert, director) through a grant from WoWiWe Instruction Co LLC.

DEDICATION

To Arvo, Duane, Rauha, Shirley.

PREFACE

WoWiWe Instruction Co was formed in 2002. While working for WoWiWe in 2010 I was tasked with "the world builder". During development, Ben Dischinger completed his masters thesis and delivered a rewritten JavaMOO.

Later, WoWiWe acquired some of WWWIC's assets for commercialization, among which was the Geology Explorer and the Virtual Cell. WoWiWe was developing the next version of the Virtual Cell, using JavaMOO to create the game world and simulations. During early testing, we noted that only a handful of players could connect to a single JavaMOO server and began to look for ways to allow for more players to connect.

Upon consultation with Dr. Harold Chaput, a WWWIC fellow and Technical Director at BioWare/Electronic Arts, we decided to pursue a RESTful (REpresentational State Transfer) architecture for the communication of game state to players. This approach promised to support more players using many servers working together; using the same technology that allows high-traffic websites to handle many users during peak demand.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

API	Application Programming Interface
HTML	Hyper Text Markup Language
HTTP	Hyper Text Transfer Protocol
JSON	JavaScript Object Notation
MBET	Model Based Exploratory Testing
MOO	MUD Object Oriented
MUD	Multi User Dungeon
REST	REpresentational State Transfer
ROA	Resource-Oriented Architecture
RPC	Remote Procedure Call
SOA	Service-Oriented Architecture
SOAP	Simple Object Access Protocol
UML	Unified Modeling Language
URI	Uniform Resource Identifier
WSDL	Web Services Description Language
WWW	World Wide Web
XML	eXtensible Markup Language

CHAPTER 1. INTRODUCTION

In our investigation of the architecture of the WWW, we look to the techniques used in the REpresentational State Transfer (REST) architecture [18]. We describe a RESTful JavaMOO, and to do so, we describe MOOs in terms of hypermedia.

The motivation for emulating WWW technologies comes as a direct result of the WWW's ability to handle large spikes in demand. We aim to reuse such load-balancing technologies to enable JavaMOO to scale effectively to handle the load generated by many connected players.

1.1. Virtual Environments

Virtual environments can transport you to places you could not otherwise visit: back in time, inside microscopic cells, or even into outer space. You can take on roles you are comfortable in, or you can try a new one: leader, teammate, small-business owner, geologist, or archaeologist [25].

The World Wide Web Instructional Committee (WWWIC) at North Dakota State University (NDSU) is an ad hoc committee of faculty, staff, and students working to advance education through the use of Immersive Virtual Environments (IVEs) [9, 10]. WWWIC, and later, WoWiWe Instruction Co (a commercial spin-off of WWWIC), have decades of experience in virtual environments [Table 1].

All of these virtual environments were created using the MOO design pattern, where the environment consists of rooms and the objects in these rooms. The players interact with the objects in a room, and by doing so, can advance through the virtual environment or complete some other goal.

1.1.1. Simulations

In these virtual environments, keys that open doors and other types of objects for completing puzzles are common themes. With some imagination, complex in-

Table 1. Listing of WWWIC/WoWiWe virtual environments.

Virtual Environment	Description
Dollar Bay [7]	An economic simulation. Players manage a small business.
Geology Explorer [2]	Players take on the role of a geologist exploring the planet Oit.
Virtual Cell [1, 28]	A biology simulation. Players explore living plant cells.
ProgrammingLand [13]	Players travel through a virtual museum with exhibits and lessons on MOOs and C++.
Like-a-Fishhook [24]	Players engage in geological, botanical and archaeological tasks at a historical site.
On-a-Slant [20]	An anthropology simulation where players travel through time to a Native American village.
Blackwood [8]	Players take on the role of homesteaders in the 1800s.
eGEO [6]	An environmental simulation. Players measure water quality on a troubled river.
FarmSim [22]	A farming simulation. Players must research and purchase seed to plant on their farm.
CIRCLE [11]	(Collaborative Identification, Retrieval, and Classification Learning Environment) in which players collect multimedia observations, then attempt to taxonomize them as a group.

teractions between multiple objects could, in essence, simulate or model real-world phenomena such as simple economic models or chemical reactions.

We have written a number of simulations for our virtual environments. Following are examples of simulations in Dollar Bay [7], Virtual Cell [1, 28], and eGEO [6]:

Dollar Bay. An economic simulation where players must manage a store. The player's store is placed in one of six areas, and the player must interpret their neighborhood's demographic information in order to stock inventory and set prices that will turn a profit. An agent-based simulation where a micro-economy is implemented by software agents that shop for virtual products.

Virtual Cell. A biology simulation in which players navigate submarines through a 3D representation of plant cells. Players interact with simulations of the electron transport chain, photosynthesis, and osmosis.

eGEO. An environmental simulation where players use various scientific tools to measure the water quality at many locations along a river. Players must interpret these measurements to find the sources of contamination in a number of scenarios.

1.1.2. Agents

Objects that move between rooms of their own accord and otherwise behave like a player are known as agents or Non-Player Characters (NPCs). These can be as simple as "conversation-bots", agents that respond with a small set of dialogue when spoken to; or complex oppositional agents with intricate behaviors that challenge the player to some sort of contest. Following are examples of agents in Dollar Bay [7] and Geology Explorer [2].

Dollar Bay. Agents in Dollar Bay include shoppers, employees, and "atmosphere" agents. Shoppers move from store to store, searching for the best deals on their "shopping lists", but are reluctant to travel far from their homes. Employees engage the shoppers when they enter the store, and handle any purchases or returns. Atmosphere agents serve as thematic entertainment for the player, entering the stores at random and offering any number of services: some beneficial and others negative.

Geology Explorer. In the Geology Explorer, agents include guides and tutors. There are various guides throughout the many regions of the game that provide the player with a sense of direction when searching for the location of assigned tasks. Tutor agents keep track of the players' progress through their tasks and give students both positive and negative feedback as they accomplish these tasks.

1.2. Multi-User Dungeons

In order to ease implementation of virtual environments the general-purpose virtual environment creator LambdaMOO [Section 1.2.2] was employed. Later, JavaMOO [Section 1.2.3] was developed to create virtual environments, as well as provide support for design guidelines previously set forth by WWWIC [29]:

- Collaborative and multi-user
- Constructed with the help of content experts
- Strong cognitive and pedagogical features to assist in learning the content
- Consistent evaluation of learning outcomes

1.2.1. MUD & MOO

Multi-User Dungeons (MUDs) are a genre of text-based adventure game. Before MUD was a genre, it was a game called "MUD" created at Essex University in 1979 [4]. Often touted as the first interactive multiplayer game, MUD was a text-based adventure game that allowed multiple players to work together [Figure 1]. As more MUDs were developed, the "MUD design pattern" came into being: rooms, exits between rooms, and verbs to interact with players and other entities.

MUD, Object-Oriented (MOO) [19] (c. 1990) extends the MUD design pattern, treating all entities in the virtual environment as objects in a class hierarchy. The primary aim of the developer is to create objects (nouns), such as `animal`, then define all of the actions (verbs) for this object, such as `give` or `put`. When subclasses of these objects are made, they automatically inherit all of the code (verbs) written for the parent object.

1.2.2. LambdaMOO

LambdaMOO [Figure 2] was used to create many of the virtual environments listed in Table 1. This was due to LambdaMOO's most compelling feature: the ability

```
> look
You are in a clearing. Exits are to the East, South, and Northwest.
> search
You search the clearing. You find nothing.
> east
You travel east on a well-beaten dirt path. You come upon a stream.
stevedore, floop, and mathy are here.
> say "hello"
player says "hello"
stevedore says "do you know how to cross the stream?"
```

Figure 1. A typical MUD session.

to modify the MOO while the server was running, allowing for instant feedback while creating objects in the MOO [14].

Programming in LambdaMOO uses a verb-based programming language where the developer defines verbs on objects, such as `describe` or `put`. The player interacts with the MOO using natural language: `put hat on cat` would invoke the `put` verb on the `cat` object, with the parameters `on, hat`.

Objects in LambdaMOO use prototype inheritance; objects are copied (using the `create` verb) as children that inherit the properties and verbs of the parent. Furthermore, any changes made to the parent object are immediately reflected in all children. e.g. if one has a `GenericHat`, and wanted to make a subclass called `FancyHat`, they would call the `create` verb on `GenericHat` and then set, say, the color to black of the resultant child object. If later it was decided that all hats need to support the `wear` verb, one would define `wear` on `GenericHat`. `FancyHat` would then automatically gain the `wear` verb as well.

While many of the virtual environments in Table 1 were initially implemented using LambdaMOO, we quickly found runtime performance degraded with many concurrent players [26]. After many years of using LambdaMOO, a list of inefficiencies began to accumulate [9]:

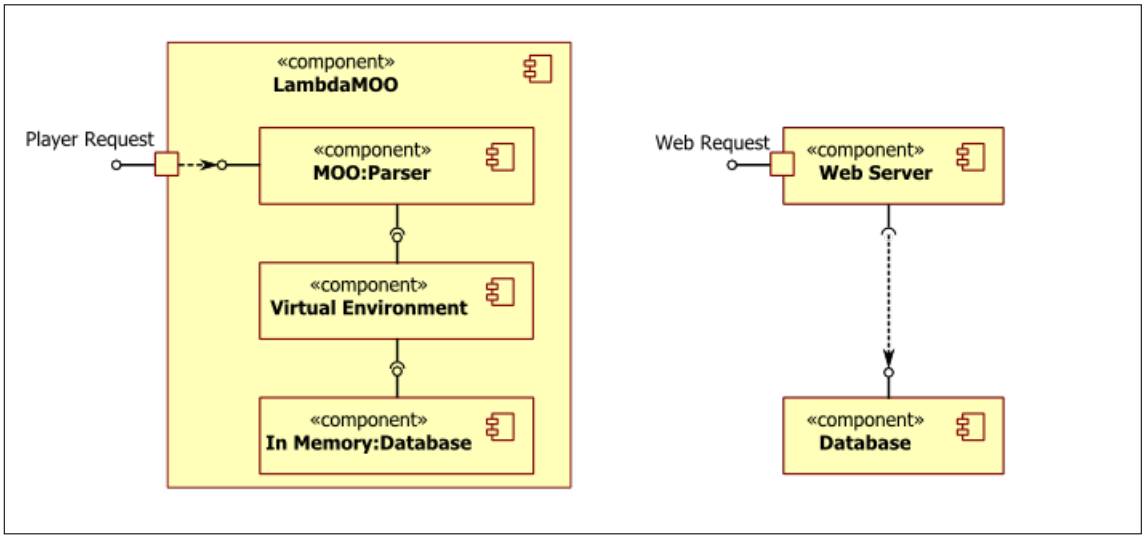

Figure 2. Component diagram of LambdaMOO's architecture. A player's actions are entered in natural language, parsed by LambdaMOO, and applied to the virtual environment. The virtual environment keeps its state in main memory. An external web server and external database serve web content.

Tick Limit. The tick limit is the mechanism LambdaMOO uses to ensure no object enters an infinite loop. Each executed instruction would increase an objects tick limit, and after a threshold was reached, the object's instructions were terminated. The tick limit made simulations and agents difficult. The programmer needed to use complicated programming techniques else the process would be killed. These same techniques to work around the tick-limit could also result in run-away processes.

Single-Thread. Execution is serial and in a single thread. Support for multiple processors or threading the execution of LambdaMOO commands is unimplemented.

Text Only. The only supported delivery mechanism is text. Graphical clients must convert player input (mouse clicks, pointer movements) into textual commands (verbs) that LambdaMOO can process. Binary messages (such as compressed text or images) could not be sent unless first converted to text.

No Version Control. When programming in LambdaMOO, the programmer typically connects to the server and begins to create objects. Whenever a programmer revises the code stored on an object, the previous code for that object is overwritten. In practice, older versions of an object's code are not saved for later review. If a programmer wants to try a different approach to a problem, they must save the text of their source code, the objects the code is on, and the relationship between those objects in some external format, or make an archival copy of the database.

External Dependencies. In order to deliver non-text content to the player, a web server must be maintained and hyperlinks given to the client. In order to store data in an application-agnostic format, an external database server also needed to be maintained.

Memory Resident. LambdaMOO is memory resident; all objects in the database are stored in main memory and the entirety of main memory is periodically check-pointed to a database file. If a new object is created and no memory can be allocated, LambdaMOO will crash.

1.2.3. JavaMOO

JavaMOO [3, 16] is a project of WWWIC (later WoWiWe) to create virtual environments using the MOO design pattern. JavaMOO is programmed using the Java language, a more modern programming language that has extra functionality to better support simulations and agents [Figure 3].

Caveat Emptor. While JavaMOO addresses the list of LambdaMOO inefficiencies [Section 1.2.2], it introduces a few constraints:

No Native Prototype Inheritance. During the initial design of JavaMOO, translation of existing LambdaMOO code was discussed at length. Java natively uses

a static class hierarchy. It was discovered that prototype inheritance would need to be emulated in Java in order to reuse existing LambdaMOO code. This was deemed infeasible to implement [16].

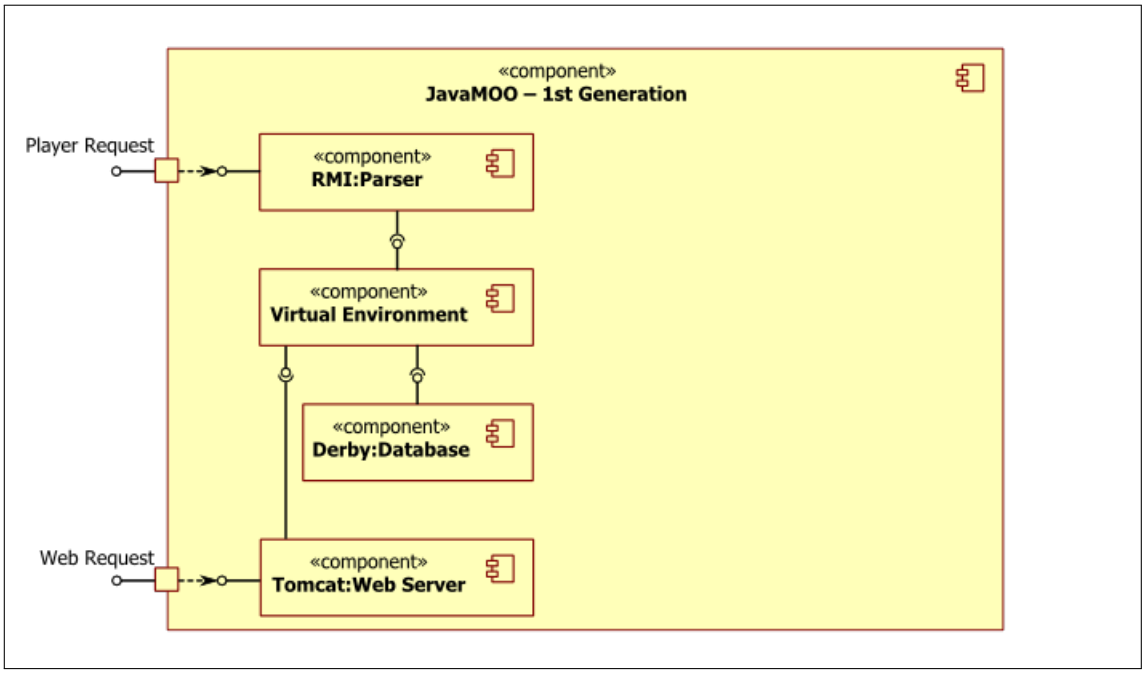

Figure 3. Component diagram of the 1st generation JavaMOO's architecture. The player's actions are sent from a graphical client (not pictured) using Java RMI (Remote Method Invocation) and executed in the virtual environment. The state of the virtual environment is backed by a Java Derby database. A Java Tomcat web server serves web content and has direct access to the virtual environment. This direct access affords the construction of interactive webpages, such as user registration or user learning assessments.

Statically Compiled. LambdaMOO environments are written in an interpreted language: the objects in the MOO could be programmed while the server was running. The use of the Java programming language restricts JavaMOO to static object definitions that cannot easily be changed at runtime.

Remote Method Invocation. JavaMOO uses the Remote Method Invocation (RMI) Java libraries to send messages between JavaMOO and its clients. This unfortunately made clients difficult to write in other programming languages.

The RMI messages closely resemble the style of message sent through a Service-Oriented Architecture (SOA) [Section 2.2.1]. Verbs are defined on the JavaMOO server (much like they are in LambdaMOO [Section 1.2.2]) and clients send messages that invoke the verbs on the server.

2nd Generation JavaMOO. After exploring the design space with the prototype implementation, JavaMOO was rewritten [Figure 4] to improve object persistence, agent control, system configuration, and otherwise follow Java best practices [16]. We found that JavaMOO could support more players in virtual environments than LambdaMOO, due to the availability of threading and non-text messaging.

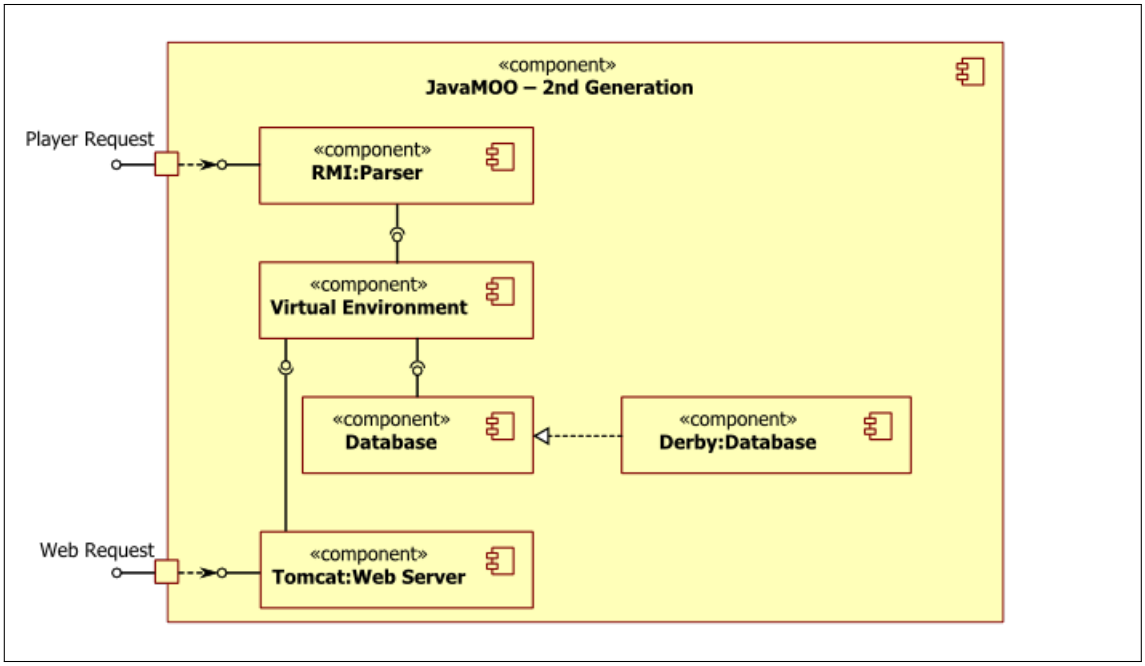

Figure 4. Component diagram of the 2nd generation JavaMOO's architecture. While the structure is similar to the 1st generation JavaMOO [Figure 3], the entire codebase was rewritten. An XML (eXtensible Markup Language) configuration system (not pictured) and an abstracted database interface were added. Java Derby remains the only realized database.

1.3. Goals & Constraints

JavaMOO proved an improvement over LambdaMOO, supporting a large-sized classroom of students. Yet assigning an entire server to each classroom of students was deemed wasteful. We decided more players needed to be supported.

Two improvements to JavaMOO are explored in this thesis: allowing MOO clients to be written in graphically-oriented software packages and the use of a distributed architecture to enable the server to better handle player load. We seek to fulfill a number goals:

G1 Design a distributed architecture

G2 Construct distributed MOOs

G3 Reduce per-player server computational load

G4 Transfer computational load to clients

In particular, a RESTful common interface [Section 3.1] is used to fulfill Goals 1, 2 and non-authoritative state synchronization [Section 3.2] to fulfill Goals 3 & 4. The use of these two design patterns alongside the MOO design pattern raise the following research questions: Does a RESTful common interface and non-authoritative state transfer enable JavaMOO to support:

- More concurrently connected players than in previous iterations?
- Complex clients written using a graphically-oriented software package?

CHAPTER 2. LITERATURE REVIEW

The techniques for distributed computing have been investigated and refined from the time that computers were first connected via a network. As these techniques matured, sets of commonly-used techniques have been recognized and organized into architectures for building distributed software. Here we examine the techniques that have enabled distributed virtual environments both in the present and the past. We also provide an overview of distributed Web technologies that are relevant to the design of JavaMOO.

2.1. Distributed Virtual Environments

Virtual environments in which multiple players can interact with each other and their environment demand enormous computing resources. Dependent on the virtual environment's requirements, the processing time or network bandwidth (or a combination of the two) of a server will be exhausted as the number of active players (i.e. the "player load") increases.

In this section we describe a number of architectures for distributing the player load across many servers, and provide examples of their implementations in commercial products and research projects.

Partition the Players. By dividing the players among many copies of the virtual environment (often called "sharding" [41]), the player load on any one server is known, or at least predictable. In the event a new player wants to join, and all servers are full, a new server is created to house the new player. In practice, new servers are created in advance and populated with many players (or existing players are migrated to the new server) to ensure a responsive environment.

Sharded database architectures are a type of shared nothing database architecture. In a shared nothing database architecture, the records (rows) in a table are split

among a number of independent servers such that neither disk space or memory are shared among them. In terms of multiplayer virtual environments, the player account and inventory tables would be sharded among all of the virtual environment servers such that a player can only connect to the shard that holds their account information.

As a result, players are isolated from other players on different servers. In the case a player wants to join friends on another server, typically a new avatar must be created or the player must obtain permission to migrate their player account [5, 38].

Partition the Services. Another distribution scheme is to partition the services needed to interact with the virtual environment. This technique allows a client to connect only to the services it requires, and also allows a client to connect to alternative service providers should their primary provider be unreachable.

OpenSimulator [30] uses service partitioning to distribute its player load. The service partitions, referred to as "UGAIM", are as follows:

User – Player authentication. With authentication provided as a separate service, web clients can be implemented that allow the player to manage their avatar, e.g. on a low-bandwidth connection without needing to connect to the other higher-bandwidth services.

Grid – Virtual environment region simulation. By defining "regions" in the virtual environment, a sense of locality can be afforded for the player, such as interacting with nearby objects or players. This service may itself be distributed, see "Partition the Environment's Regions" below.

Asset – Media management. Music, images, avatars and other 3D models are typically cacheable static resources that are most efficiently considered separately from other, mutable environment state.

Inventory – Game state management. The state of the player's account and avatar, including any game objects or other mutable state such as the player's position and the player's current region.

Messaging – Inter-player communication. Often a player will want to communicate with friends, who may be in a nearby region, far away, or even offline. Thus, messages are handled as a service separate from the "Grid" service.

Partition the Environment's Regions. This is by far the most popular method to distribute a virtual environment. In a large virtual environment players often do not interact over large distances [23]. Due to the lack of cross-region interaction, these regions in the virtual environment can be distributed across many servers, either by some static definition (as in Second Life [34]) or dynamically based on player load [17, 23].

EVE Online [12], a large multiuser environment, uses region-based clustering. EVE Online follows the MUD design pattern; its regions are a hierarchal collection of connected rooms. Each top-level room or "sector" is executed on its own server, along with its children rooms.

2.1.1. Distributed MUDs

MUDs have had support for "portals" or "cyberportals" since 1990 [33]. These portals allowed a player to exit a room in their current MUD and enter a room on another MUD, provided the player had an account on the destination MUD. With the release of UnterMUD [37] in 1992, players were able to migrate their player's mutable state when using a portal, but only if the destination MUD was also an UnterMUD.

CoolMUD [36], released in 1993, was the first MUD to provide portals that did not require client support (previous implementations of portals were performed by the player's client). CoolMUD also had support for distributed objects. When

a player interacted with an object that was on another server, that command was transparently routed to the object's server to be executed there.

2.2. Web-Based Architectures

Originated in 1991, the WWW was designed to support a distributed information system. Webpages written in "hypertext" (text containing "hyperlinks" – links to other webpages) allowed authors to reference information on other network-connected computers, a technology that quickly grew into the web as we know it today.

The web was at first a static repository of hyperlinked information. Today's web also includes dynamic content, created and updated on demand. One type of dynamic webpage, a "web application", aims to provide the functionality of a regular computer application over the Internet.

In essence, applications can be abstracted to the state of the application, and the possible operations on that state. The state, or data, of the application can be thought of in terms of *nouns*: people, places, accounts, and other objects. Likewise, the operations on these state can be described as *verbs*: hire, describe, calculate, and other actions.

In this section, we will review two architectures for creating web applications: Service-Oriented Architectures (SOA), which are primarily concerned with verbs; and Resource-Oriented Architectures (ROA), dealing in nouns.

2.2.1. Service-Oriented Architecture

In an SOA, the developer is concerned with the creation of verbs (operations on data) and mapping these verbs to an Application Programming Interface (API) [21, 32]. The user is presented with a view of the application while the application state is stored and manipulated on the server.

This type of architecture is the most popular of web-based architectures due to the ease and speed of implementation [21]. An SOA is a natural extension of the

Object-Oriented design pattern, where data and the functions to process the data are coupled as a single object. In the case of an SOA, the objects containing the data are stored on a remote server where the processing of the data also occurs.

SOAP. The Simple Object Access Protocol (SOAP) [42] is the primary method used to send messages between SOA web applications and the services they depend on [32]. SOAP uses eXtensible Markup Language (XML) to encapsulate each message.

The author of a web service writes a formal description of the functions (verbs) available and their parameters in the machine-readable Web Services Description Language (WSDL), an extension of XML. The client then accesses this WSDL file to determine how to access and use the service. Finally, an SOA client sends SOAP messages with the verbs defined by the web service to perform RPCs on the application state stored on the server.

2.2.2. Resource-Oriented Architecture

An ROA concerns the developer with the mapping of nouns (data or state) to web resources that are available to clients for further processing [15]. The canonical ROA implementation is REST [18], in which nouns are mapped to URIs and presented through hypermedia to the client. Where REST is a general architecture that can be applied to different technologies, an ROA further restricts REST by requiring the use of web technologies: the HyperText Transfer Protocol (HTTP), HyperText Markup Language (HTML), and the interaction patterns used on the World Wide Web (WWW).

The verbs (data operations) are limited to those used in HTTP (most commonly `GET`, `POST`, and `DELETE`). The client interacts with the server by requesting a hypermedia page (usually an HTML page), which the server populates with all possible actions (hyperlinks) that the client can perform.

REST. The constraints for a RESTful application are:

Hypermedia Driving Application State. All actions should be visible from the client's current state. The current state should be represented as hypermedia (i.e. contain hyperlinks). This implies no outside knowledge of available actions.

Stateless Interactions. Messages between the server and the client should not rely on client state stored on the server, e.g. the server should not keep a substantial amount of client state in global memory (such as a hash map indexed by session id), preferring to represent the state in the hypermedia sent to the client (and thus keeping the client state on the client).

Resource Identification. Every resource should have a URI. This allows the server and client to reference a resource by a universal and stable name, as well as affording the use of load-balancing and caching servers in a layered network topology.

Self-Describing Messages. All the information needed to process a message is sent with each message. This supports "stateless interactions" as well as affords the use of a layered network topology.

Uniform Interface. Using a well-defined and stable uniform interface allows the client(s) and server to improve independent of each other, so long as the uniform interface remains unchanged.

2.2.3. Summary

Both SOA and ROA use the web to deliver an application to the user. An SOA keeps the data and the processing on the server. Summary information is delivered to the user through an interface (usually SOAP) and the client issues commands (RPCs) to the server to act upon the data. An ROA keeps a published version of the data

on the server. Processing of the data occurs on the client, who may then publish the updated data for others to view.

CHAPTER 3. A RESTFUL APPROACH

In order to satisfy our goals [Section 1.3] we abstract the RMI implementation to allow other message transports, such as TCP and HTTP. We also make a number of changes to JavaMOO to support the use of URIs when referencing objects. We place these additional restrictions on the JavaMOO architecture [Figure 5]:

- **Common Interface.** The client and server should communicate using a message format that can be parsed by many programming languages.
- **Non-Authoritative State Synchronization.** The server should not try to calculate the full state of the simulation, especially the size and shape of players and their collisions.

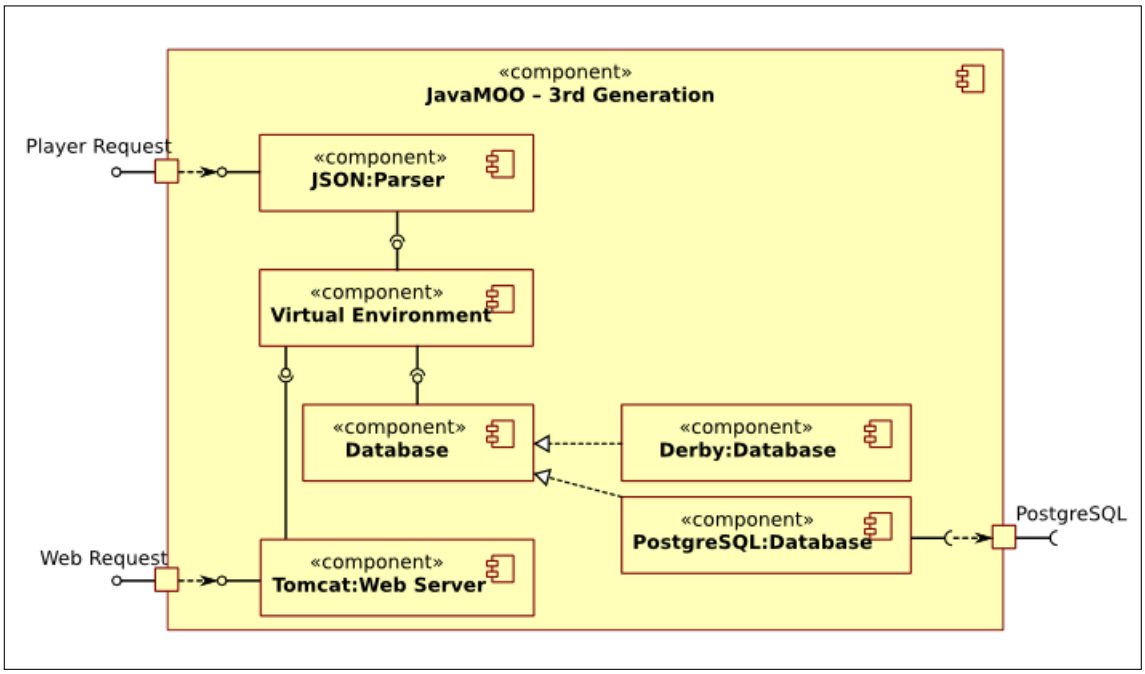

Figure 5. Component diagram of (the 3rd generation) JavaMOO's architecture. A direct extension of the 2nd generation JavaMOO [Figure 4] codebase. Two improvements are depicted: the RMI parser is replaced with a JSON parser and a second database realization is added: PostgreSQL.

3.1. Common Interface

In order to afford a flexible client implementation, the client and server should communicate using a message format that can be parsed by many programming languages. The use of a common interface also affords separate client and server development, resulting in a server that can talk to many versions of client and clients that can talk to many versions of server, a departure from previous experiences with JavaMOO and RMI.

Previous experience with Java Remote Method Invocation (RMI), a type of RPC, was not satisfactory. The use of RMI made the construction of non-Java clients infeasible. Previous implementations of JavaMOO placed the communication logic in the server's namespace, such that the Java clients used RMI to execute code on the server to queue their commands. Java clients quickly became difficult to follow; one had to carefully trace execution of commands that were interleaved on the server and client in order to find the underlying causes of bugs.

The use of a common interface, such as JSON [27] (JavaScript Object Notation, a "lightweight data-interchange format") over HTTP, affords writing clients in a programming language different than that of the server, typically a programming language more suited for graphical clients as opposed to the programming languages typically used to write servers.

3.2. Non-Authoritative State Synchronization

How do we keep many players in a shared view of the environment, especially when the environment's state might update many times a second? Initial attempts to simulate state were implemented exclusively on the server, with each client merely viewing the simulation, much like the OpenSimulator architecture [30]. This places untenable load on a server when high definition/high resolution 3D graphical objects are needed for the sake of authentic visualization.

A non-authoritative approach [40] to the state synchronization problem was taken to compensate for these constraints. This approach places handling of player input and event interactions locally on each client. The final results are then sent to the server where each clients' input is mediated and the environment state updated.

3.2.1. REST

After many design meetings and consultations, we decided to pursue a RESTful [18] server. The use of a RESTful API to transmit environment state provides scalability for the virtual environment, allowing three times the number of concurrently connected players than our previous attempts [26]. This scalability is afforded by the reuse of the load balancing technologies that power the WWW. Layered server architectures and caching are afforded by the use of URIs.

Hypermedia. We must first make small changes so that the MOO environment can be described as hypermedia [Figure 6]. We begin by defining all commands (verbs) as being in one of four categories:

- `GET` Retrieve/describe an existing object.
- `POST` Create a new object.
- `PUT` Update an existing object.
- `DELETE` Delete an existing object.

In practice, `DELETE`-type commands need not be defined or be accessible to players and `PUT`-type commands can be implemented as `POST`-type commands that either create a new object, or if one exists, overwrite that object, leaving two categories:

- `GET` Retrieve/describe an existing object.
- `POST` Create a new object, or overwrite an existing object.

Client: `GET /` The client enters the MOO from a bookmark or other hyperlink.

Server: `HTTP 307 Temporary Redirect; Location /moo/lobby/` The server redirects the client to the "lobby" object, the entry point to this MOO.

Client: `GET /moo/lobby/` The client automatically follows the `307 Temporary Redirect`.

Server: `HTTP 200 OK` The server returns a hypermedia description of the "lobby" object, which happens to be room.

Client: `GET /moo/lobby/door/` The player reviews the "lobby" object's description. The player decides to inspect one of the objects in the lobby, a "door", by following the provided hyperlink.

Server: `HTTP 200 OK` The server returns a hypermedia description of the `door` object, which is an exit to another room, "lounge".

Client: `GET /moo/lounge/` The player decides to follow the hyperlink retrieved from the `door`'s description.

Server: `HTTP 200 OK` The server returns a hypermedia description of the "lounge" object, which is another room.

Client: `POST /moo/lounge/chair/` The player creates an object in the "lounge" room, using a hyperlink provided in the `/moo/lounge/` description (e.g. a `<form>`-type element if using HTML).

Server: `HTTP 303 See Other; Location: /moo/lounge/chair/` The server informs the client that the result of the `POST` action can be seen at `/moo/lounge/chair/`.

Client: `GET /moo/lounge/chair/` The client automatically follows the `303 See Other`.

Server: `HTTP 200 OK` The server returns a hypermedia description of the "chair", a new player-created object.

Figure 6. Example HTTP Interaction with a RESTful MOO. Player authentication is ignored.

3.3. RESTful Simulations

In a virtual environment that lacks simulations, the state of the environment is determined solely by player actions. Following the REST model, there is a clear separation between retrieval and mutation of state. Many requests to retrieve state from any number of players will return identical information. Only when a player updates the environment state does the environment change. When we add simulations to our virtual environments we are faced with many alternative implementations, among which are the use of agents and the use of "reentrant" simulations.

Implement Using Agents. An agent can emulate a player, making changes to the environment state using the same mechanisms a player would. This solution is difficult to implement, as many internal details need to be made available through the API in addition to the complex logic that must be written for the agent.

Implement Using Reentrant Simulations. The simulation can update only when a "relevant" action is performed by a player, as a part of processing the result of the player's action. By "reentrant" simulation, we mean a simulation that can update its state multiple times a second just as well as over a time span of days or longer. The use of internal buffers or queues is not desirable, as they may eventually overflow if no players interact with the simulation for too long. Instead algorithms must be written that can update the state of the simulation without relying on a constant update interval. This technique has a useful side-effect: if nobody is interacting with a simulation, no processing of the simulation takes place.

CHAPTER 4. THE VIRTUAL CELL

No plan survives contact with the enemy.

– Helmuth Graf von Moltke, *Militarische Werke II, part 2*, 1900

They say no plan survives first contact with implementation.

– Andy Weir, *The Martian*, 2014

Virtual Cell [28], released by WoWiWe Instruction Co in 2013, is an immersive three-dimensional environment for teaching the processes and structures of Cellular Biology. It currently contains real-time simulations of the electron transport chain, the photosynthesis light reactions, and osmosis.

The JavaMOO server keeps track of positions of players and items in the context of each room, while the client renders a scene of 3D objects. All player actions are visible to every other player in the same room. The client-server interaction can be modeled as a 3D hypermedia client and non-HTTP hypermedia server.

4.1. Client

The Virtual Cell client was implemented in Unity3D [39] using the C# programming language. In order to communicate with the Virtual Cell server (written using JavaMOO, which at the time only provided RMI communication) we began an iterative approach to using RESTful techniques during construction of the Virtual Cell server [Figure 7].

While REST is typically implemented using HTTP to transfer messages, we decided to forgo the complexity of a complete HTTP implementation in the first iteration. We opted to construct a TCP-socket implementation, as an HTTP implementation can later be constructed on top and most programming languages support communication over a socket interface.

Non-Authoritative State synchronization. In order to keep most of the 3D scene processing on the client, we implemented a matching algorithm [Figure 8] that allows the client to pre-place objects then later match them to server URIs.

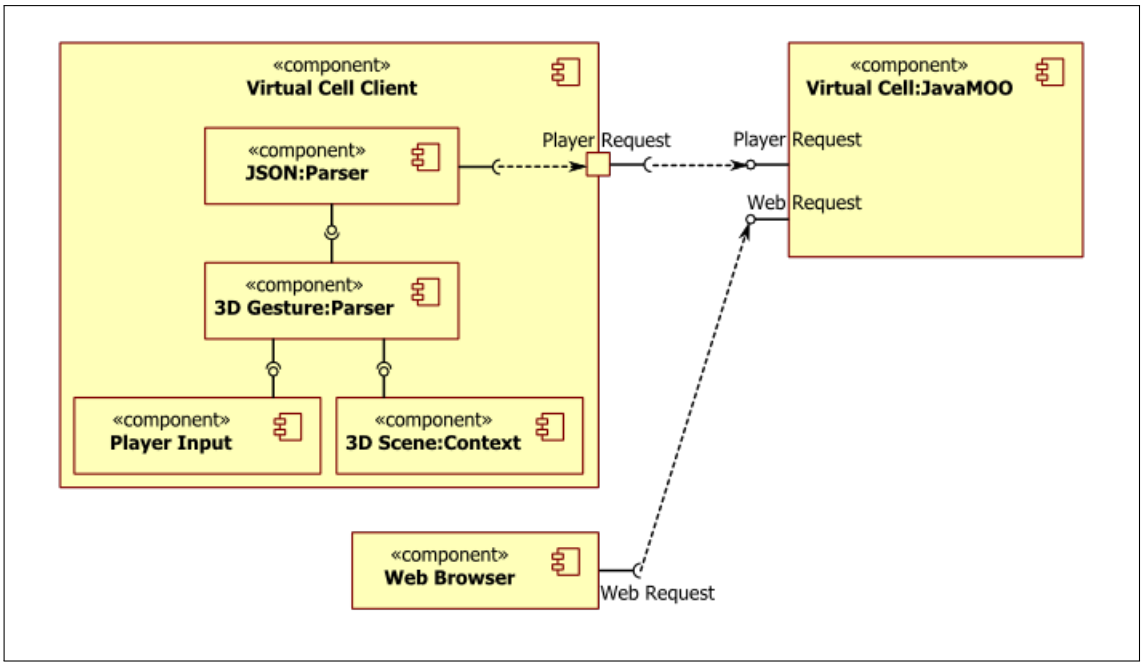

Figure 7. Component diagram of the Virtual Cell. Depicted is the interaction of the Virtual Cell JavaMOO server and the Virtual Cell client. The client interprets the player's input in the context of the 3D scene then sends the player's actions over the JSON common interface to the server for mediation and dissemination to other clients. An external web browser is used for user registration.

This enabled handling of player input, calculation of physics, and event interactions to occur locally on each client. Additionally, the server specifies which 3D assets should be loaded for each URI in the object's description, allowing the client to perform animations or load, position, and display 3D models without relying on the server to update the states of those animations and models.

In order for the server to keep player actions synchronized across multiple clients, the simulation interaction algorithm [Figure 9] was used. Such techniques worked well unless the player left the simulation. In the end, an agent was written that toured the rooms and removed leftover objects that were abandoned by players.

Client: ChangeRoom. The player enters a room.

Server: The contents of the previous and current rooms are updated.

Client: A predefined 3D scene containing pre-placed game and non-game objects is loaded.

Client: GetRoomContents. The player requests the contents of the room.

Server: A list of game objects in the room, and their properties (including a URI for each object) is returned.

Client: The client reconciles its view with the server's:

- The list of predefined game objects in the scene is sorted by type and by distance from the origin point (0, 0, 0).
- The list of URIs returned from the server is sorted by type and alphabetized.
- The lists are merged.
- Any remaining URIs are either assigned to newly-instanced game objects or discarded.
- Game objects in the scene with no assigned URI are disabled.

Figure 8. Matching algorithm used when rendering scenes. The client has a predefined scene of 3D objects, and must pair each object with a URI from the server in order to communicate the object's state.

4.2. API

To support client and server interaction, a common communication interface must be developed. JSON [27] was selected because it provides a programming language agnostic messaging format that has library support in many major languages.

Common Interface. The API [Figure 10, Appendix B] is implemented using JSON for the common interface between client and server. Some of the player's mutable state variables can be set directly (players' position and room) while others need to be set indirectly (goal state and cell health) as side-effects of the direct-state setters.

Client: Each game object in the scene is associated with a URI obtained in [Figure 8].

Client: GetInventory. The player requests their inventory.

Server: A list of game objects in the player's inventory, and their properties (including a URI for each object) is returned.

Client: FireItem. The player requests a game object (via its URI) be placed in the room.

Server: The game object:

- Is removed from the player's inventory
- Maintains its ownership field
- Is returned along with its properties, including a (possibly new) URI

Client: GetSceneState. Meanwhile, the client has been refreshing its view of the simulation.

Server: Players are informed a new game object has been added to the room.

Client: If the game object is of a predefined type, a new instance is created in the scene with the properties given by the server.

Client: UseItem. Meanwhile, clients are checking for collisions of game objects. When a collision occurs, the URIs of the colliding game objects are sent.

Server: The ownership field is checked against the callee. If they match, the simulation state is updated.

Figure 9. Simulation Interaction Algorithm. Any number of clients may observe a player interacting with objects in the room. The server maintains consistent state all clients.

Client: {"type":"Login", "username":"matti", "password":"abc"}

Server: {"type":"LoginResponse", "token":"matti1393004666204"}

Client: {"type":"GetInventory", "token":"matti1393004666204"}

Server: {"type":"GetInventoryResponse", "items":[]}

Figure 10. Example Virtual Cell API Message Exchange. The player logs in, then requests a description of their inventory state.

4.3. Simulations

In pursuit of non-authoritative state synchronization, we moved all of the simulation's animations to the client. We then devised methods [Figure 8, 9] to enable the server to inform the client of any animation updates through the descriptions of the objects in the room.

Electron Transport Chain Simulation. In this simulation, many copies of the electron transport chain (ETC) are placed in the cristae of a mitochondrion [Figure 11]. For pedagogical purposes, the complexes that compose the ETC are placed in linear order of their function.

Players observe the complexes as they move protons and electrons through the ETC. Each complex is a separate game object. The server keeps a list of complexes in the mitochondrion and sends the list to the client. The client uses a matching algorithm [Figure 8] to decide how many ETCs should be shown and their placement such that their positions are consistent across all clients.

Players then perform tests called assays and attempt repairs by shooting objects at the complexes. The client informs the server if a collision occurs, and the server returns the new game state [Figure 9].

The objective of this simulation is to watch and test for the complex in each chain that is failing to perform its function. The player watches as electrons are

Figure 11. Electron Transport Chain (ETC) simulation. In the foreground are two ETCs embedded in the interior of a mitochondrion and a collaborator (submarine, lower-left quadrant). Players examine, probe, diagnose, and finally repair any faulty complexes encountered.

moved through the ETC. At some point, a complex can be seen to fail in its function. The player shoots various substrates into the ETC, which may jump-start the ETC for a cycle or two before failing again. The player must use diagnostic reasoning to discover which complex is faulty, and after fixing many ETCs the player learns to associate various substrates with the complexes that are affected by them. The player then finds a replacement complex and places it into its position in the chain.

Photosynthesis Simulation. Modeling of photosynthesis reactions is much the same as our ETC simulation. The complexes involved in photosynthesis are again placed linearly, this time embedded in the thylakoid membrane inside a chloroplast [Figure 12]. Photons striking the photosystem complexes are depicted as red streaks of light. The same matching algorithm [Figure 8] used in the ETC allows the player to interact with the photosynthesis complexes.

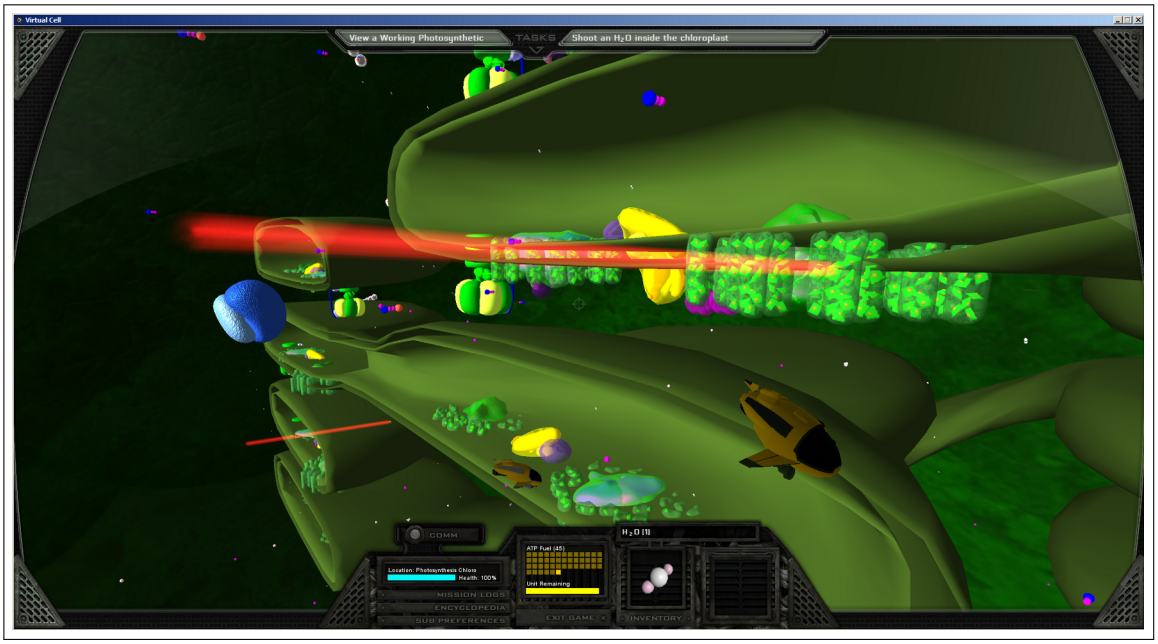

Figure 12. Photosynthesis simulation. The player is inside a chloroplast, along with two other players (submarines). Two photons (red streaks of light) are about to strike two photosystem complexes. Players must diagnose the missing component in the chloroplast as a whole. For example, if too little light is present, photon torpedoes are deployed to stimulate the reaction.

While the ETC simulation deals with each set of complexes independently, allowing players to replace damaged complexes, the photosynthesis simulation requires players to identify missing components within the entire simulation. To model this, we introduce tests that are performed on the chloroplast room the complexes inhabit. The players then drop the missing components (water, light, carbon dioxide, or catalytic proteins) into the room in order to repair the entire chloroplast.

Osmosis Simulation. The ETC and photosynthesis modules simulate processes inside cell substructures. The osmosis simulation models a still larger system: the flow of water in the entire cell. An animation of a shrinking cell with blockages (that prevent the flow of water) was implemented on the client [Figure 13]. As the player repairs the problem, the server updates a numeric "health" for the cell. The client uses this health to update the shrinking cell animation.

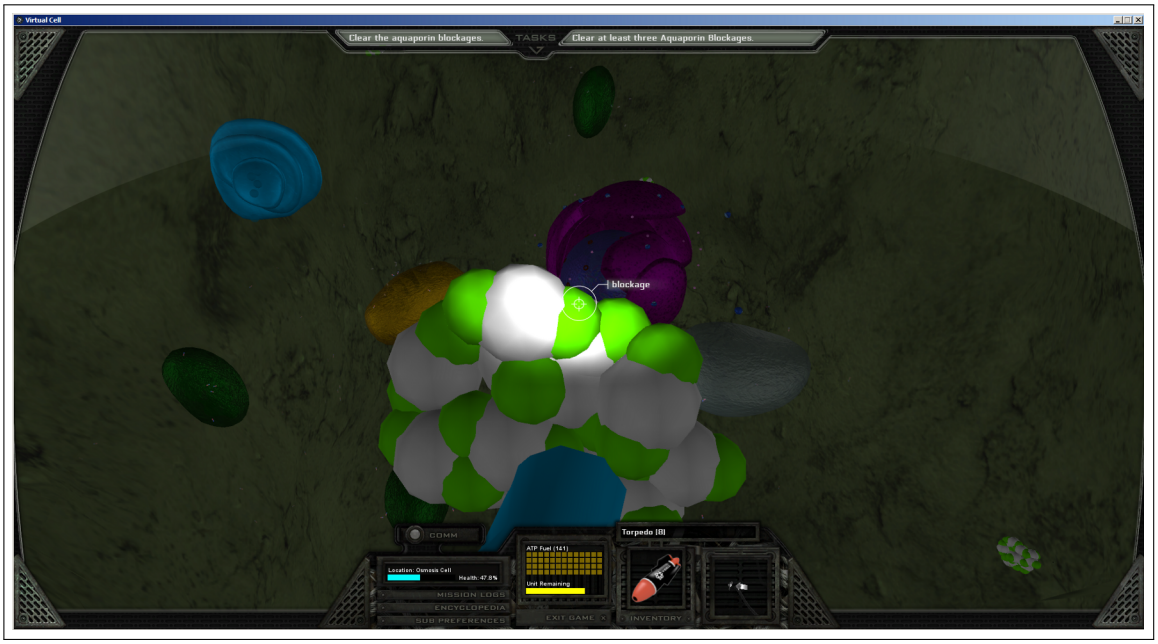

Figure 13. Osmosis simulation. The player is examining a plasmodesma blockage (foreground) which is preventing water from flowing into the cell (background). The player must fire concussive torpedoes to remove the blockage.

In order to calculate the health of the cell, the server monitors the contents of the cell room as well as a subset of the rooms that link to it (rooms that represent the inner contents of objects in the cell). Players use the tests developed for the photosynthesis simulation to discover which substance needs to be dropped in which room to improve the health of the cell.

4.4. Agents

The tutoring (guidance and remediation) in the Virtual Cell [31] uses a goal system adaptable to players of all abilities. The tutoring is located entirely on the server. Players are assigned tasks and guidance in accordance with their learning goals.

Individualized tutoring is provided to players who are unable to complete their tasks. The player's history is recorded on the server, and later searched for patterns indicative of confusion. Two types of tutoring are employed: blind and orientated

tutoring. Blind tutoring is applied when no pattern can be found, or the pattern found indicates a chance mistake. Orientated tutoring is used when a found pattern suggests the confusion of two similar topics.

4.5. Conclusion

This 3rd generation JavaMOO improves on the design of previous implementations by replacing the RMI communication layer, which promoted less-scalable RPC interactions, with a JSON common interface that allows for a layered, distributed server network and clients written in other programming languages.

Better server scalability is achieved [35, 26] in the Virtual Cell JavaMOO implementation through the use of RESTful design patterns: URIs for each game object, self-describing messages (all information needed to process each API request is contained in the request) and the JSON common interface that enables a graphically-rich 3D client. The use of non-authoritative state transfer allows the graphical client to provide a smoother multiuser experience by allowing the client to partially simulate non-critical portions of the virtual environment's state.

CHAPTER 5. FUTURE WORK

5.1. HTTP Communication Layer

A TCP communication layer was constructed for the JSON common interface with the Virtual Cell client. HTTP is typically implemented on top of TCP by appending headers of the form `key: value`, one per line, followed by an empty line and finally the hypertext body of the document. Rather than implement HTTP from scratch, the Apache libraries have been used in an experimental HTTP communication layer implementation located in the `javamoo.net.http` namespace.

This implementation lacks support for dynamic server-side programming (such as `.jsp` pages) or utility code to parse JSON from `GET`-style query strings. Sending JSON in the body of a `POST` request is supported, but the response is not the expected format of many third-party web libraries, such as jQuery.

5.2. Distributed Datastore

JavaMOO's persistence layer keeps a cache of objects fetched from the backing store in main memory. There exists no mechanism to automatically detect or otherwise programatically inform this cache if it is inconsistent with the backing store. Instead, the dirty cache is flushed to the backing store on the next write. This makes third-party utilities and other JavaMOO instances unable to modify the backing database, unless the JavaMOO server with a dirty cache is first shut down.

Calls to external databases are not automatically batched, and no mechanism is in place to manually batch database calls. This is aggravated by the class-hierarchy design pattern of the objects stored in the database: each object queues a database write at least once for each level in the class hierarchy. In addition, there is no database profiling support to make such SQL-batching decisions.

5.3. Reduce URI-Server Coupling

The URIs in JavaMOO are of the form:

```
{long_id}~{classname}~{database_url}
```

For instance:

```
35~javamoo.domain.threeD.Room3d~jdbc:derby:maindb;create=true
```

for a local Derby database and:

```
89~javamoo.domain.Token~jdbc:sql://db.s108.com/db108-accounts/
```

for an external database. In terms of hypermedia, the current implementation uses "unaliased absolute URIs" (this object number (row), this table, this server and database). A better approach would to also have support for relative URIs, or at the very least "aliased absolute URIs" (this object number (row), this table, this named server and database) such as:

```
89~javamoo.domain.Token~jdbc:sql://login-server/
```

5.4. Identify RESTful Best Practices

RESTful messages are not enforced programmatically nor through any sort of source code analyzer. Future implementers are not prevented from violating RESTful practices, and thus losing the scalability that REST offers. A document and/or working examples of MOOs, both distributed and not, would provide junior implementers with a solid foundation to build upon.

5.5. Server Management

In addition to the lack of support for third-party database modifications, object inspection tools, object creation tools, and database check-pointing tools are needed. In particular, the ability to migrate user accounts to newer versions of the server or otherwise upgrade a server while keeping the user accounts remains to be implemented.

A user migration API event was designed, but remains to be fully implemented (user accounts refer to objects by absolute URI, and these must be linked to objects on the new server for user migration to succeed).

JadeMOO (a database inspection and object creation tool for the previous implementation of JavaMOO) has been ported to this JavaMOO implementation. Its ability to create or update an object in the database is currently unimplemented.

5.6. Layered Login Server

When players want to connect to the virtual environment, they send their `username` and `password` to their game server, receive a `token`, then begin to play the game through their client. Instead, a player could connect to a central login server, provide their credentials, then be forwarded to the correct server.

This simplifies the management of game servers: only the login server must know which servers are online and which are not. This also simplifies the configuration of clients: all clients need only point to a central login server rather than be individually configured for each game server.

REFERENCES

[1] A.R. White, P. McClean, and B.M. Slator, *The Virtual Cell: A virtual environment for learning cell biology*, Tenth International Conference on College Teaching and Learning (Jacksonville, Florida), 1999.

[2] B. Saini-Eidukat, D.P. Schwert, and B.M. Slator, *Geology Explorer: Virtual geologic mapping and interpretation*, Journal of Computers and Geosciences **27** (2001), no. 4, 1167–1176.

[3] B. Vender, O. Borchert, B. Dischinger, G. Hokanson, P. McClean, and B.M. Slator, *JavaMOO virtual cells for science learning*, Virtual Immersive and 3D Learning Spaces: Emerging Technologies and Trends, IGI Global, 2010, pp. 194–211.

[4] R.A. Bartle, *Interactive multi-player computer games*, `http://mirrors.ccs.neu.edu/MOO/papers/mudreport.txt`, 1990.

[5] Blizzard, *Character transfer FAQ*, `http//us.battle.net/support/en/article/character-transfer`, 2006, Accessed: 2014 Feb 22.

[6] B.M. Slator, B. Saini-Eidukat, D.P. Schwert, O. Borchert, G. Hokanson, S. Forness, and M. Kariluoma, *The egeo virtual world for environmental science education*, Geological Society of America Abstracts with Programs (Minneapolis, Minnesota), 2011, p. 135.

[7] B.M. Slator and G. Farooque, *The agents in an agent-based economic simulation model*, International Conference on Computer Applications in Industry And Engineering (Las Vegas, Nevada), 1998.

[8] B.M. Slator, K. Wynne, D. Burleigh, J. Kadrmas, E. Kennedy, et al., *Rushing headlong into the past: the blackwood simulation*, Proceedings of the Fifth IASTED International Conference on Internet and Multimedia Systems and Applications (Honolulu, Hawaii), 2001, pp. 318–323.

[9] B.M. Slator, O. Borchert, L. Brandt, H. Chaput, K. Erickson, G. Grosebeck, J. Halvorson, J. Hawley, G. Hokanson, D. Reetz, and B. Vender, *From dungeons to classrooms: The evolution of muds as learning environments*, Evolution of Teaching and Learning Paradigms in Intelligent Environment, Springer-Verlag, Heidelberg, Germany, 2007.

[10] B.M. Slator, R. Beckwith, L. Brandt, H. Chaput, J.T. Clark, L.M. Daniels, C. Hill, P. McClean, J. Opgrande, B. Saini-Eidukat, D.P. Schwert, B. Vender, and A.R. White, *Electric worlds in the classroom: Teaching and learning with role-based computer games*, Teachers College Press, New York, 2006.

[11] O. Borchert, *Harnessing user generated multimedia content in the creation of collaborative classification structures and retrieval learning games*, Seminar – Equity in the STEM Disciplines (Fargo, North Dakota), 2013.

[12] CCP, *My node was equipped with the following...*, `http://web.archive.org/web/20100110021300/http://www.eveonline.com/devblog.asp?a=blog&bid=589`, 2008, Accessed: 2010 Jan 10.

[13] C.D. Hill, B.M. Slator, and L.M. Daniels, *Using and validating programmingland*, Proceedings of the 7th IASTED International Conference on Computers and Advanced Technology in Education (Kauai, Hawaii), 2004, pp. 291–296.

[14] P. Curtis, *Not just a game: How LambdaMOO came to exist and what it did to get back at me*, High Wired: On the Design, Use, and Theory of Educational MOOs, University of Michigan Press, 1998.

[15] D. Guinard, V. Trifa, and E. Wilde, *A resource oriented architecture for the web of things*, Internet of Things (IOT), 2010, 2010, pp. 1–8.

[16] B. Dischinger, *An architecture for the implementation and distribution of multiuser virtual environments*, Master's thesis, NDSU, Fargo, North Dakota, 2010.

[17] D.T. Ahmed and S. Shirmohammadi, *A microcell oriented load balancing model for collaborative virtual environments*, Proceedings of the IEEE Conference on Virtual Environments, Human-Computer Interfaces and Measurement Systems, 2008, pp. 86–91.

[18] R. Fielding, *Architectural styles and the design of network-based software architectures*, Ph.D. thesis, UCI, Irvine, California, 2000.

[19] K. Fox, *MOO-Cows FAQ*, `http://www.moo.mud.org/moo-faq/moo-faq-1.html`, 2004.

[20] G. Hokanson, O. Borchert, B.M. Slator, J. Terpstra, J.T. Clark, L. M. Daniels, H.R. Anderson, A. Bergstrom, T.A. Hanson, J. Reber, D. Reetz, K.L. Weis, A.R. White, and L. Williams, *Studying native american culture in an immersive virtual environment*, IEEE International Conference on Advanced Learning Technologies (2008), 788–792.

[21] H. He, *What is service-oriented architecture?*, XML.com, O'Reilly Media, 2003.

[22] G. Hokanson, *NDSU farm simulator*, `http://eieio.cs.ndsu.nodak.edu/`, 2014.

[23] I. Kazem, D.T. Ahmed, and S. Shirmohammadi, *A visibility-driven approach to managing interest in distributed simulations with dynamic load balancing*,

Proceedings of the 11th IEEE International Symposium on Distributed Simulation and Real-Time Applications, 2007, pp. 31–38.

[24] J.T. Clark, B.M. Slator, A. Bergstrom, S. Fisher, J. Hawley, E. Johnston, J.E. Landrum III, and M. Zuroff, *Virtual archaeology as a teaching tool*, Proceedings of the 30th Conference on The Digital Heritage of Archaeology (Heraklion, Crete), 2003, pp. 29–35.

[25] L. Brandt, O. Borchert, K. Addicott, B. Cosmano, J. Hawley, G. Hokanson, D. Reetz, B. Saini-Eidukat, D.P. Schwert, B.M. Slator, and S. Tomac, *Roles, culture, and computer supported collaborative work on planet oit*, Proceedings of the International Conference on Computers and Advanced Technology in Education, 2005.

[26] M. Kariluoma, B.M. Slator, B Vender, O. Borchert, G. Hokanson, P. Yan, and B. Cosmano, *The design of multiplayer simulations for online game-based learning*, Proceedings of the IASTED Technology for Education and Learning Conference (Marina del Ray, CA), 2013.

[27] N. Nurseitov, M. Paulson, R. Reynolds, and C. Izurieta, *Comparison of JSON and XML data interchange formats: A case study*, International Conference on Computer Applications in Industry And Engineering (San Francisco, California), 2009, pp. 157–162.

[28] O. Borchert, G. Hokanson, B.M. Slator, B. Vender, P. Yan, V. Aggarwal, M. Kariluoma, A. Marry, and B. Cosmano, *A 3D immersive virtual environment for secondary biology education*, Proceedings of Society for Information Technology & Teacher Education International Conference (New Orleans, Louisiana), 2013.

[29] O. Borchert, L. Brandt, G. Hokanson, B.M. Slator, B. Vender, and E.J. Gutierrez, *Principles and signatures in serious games for science education*, Gaming and Cognition: Theories and Practice from the Learning Sciences, Information Science Publishing, 2010.

[30] OpenSimulator, *Introduction to OpenSimulator*, `http://opensimulator.org/wiki/OpenSim:Introduction_and_Definitions`, 2011.

[31] P. Yan, B.M. Slator, B. Vender, W. Jin, M. Kariluoma, O. Borchert, G. Hokanson, V. Aggarwal, B. Cosmano, K.T. Cox, A. Pilch, and A. Marry, *Intelligent tutors in immersive virtual environments*, Conference on Cognition and Exploratory Learning in Digital Age (Fort Worth, Texas), 2013.

[32] M.P. Papazoglou, *Service-oriented computing: Concepts, characteristics and directions*, Proceedings of the 4th International Conference on Web Information Systems Engineering, 2003, pp. 3–12.

[33] A. Rang, *TinyTalk 1.0 is now available for anonymous FTP*, http://groups.google.com/group/alt.mud/msg/cd7c39ab3fbefbb2, 1990, Accessed: 2014 Feb 20.

[34] M. Rymaszewski, *Second Life: The official guide*, John Wiley & Sons, New York, 2007.

[35] C.J. Schaefer, *Model-based exploratory testing: A controlled experiment*, Master's thesis, NDSU, Fargo, North Dakota, 2013.

[36] S.F. White and R.L. Powell, *The CoolMUD server*, http://vrici.digitalkingdom.org/~rlpowell/software/coolmud/, 1994, Accessed: 2014 Feb 20.

[37] J. Smith, *Untermud*, http://www.lysator.liu.se/mud/faq/servers.html, 1993, Accessed: 2014 Feb 20.

[38] Square-Enix, *World transfer service*, http://www.playonline.com/ff11us/intro/optional/wd_transfer01.html, 2002, Accessed: 2014 Feb 22.

[39] Unity Technologies, *Unity – game engine*, http://unity3d.com/, 2014.

[40] V. Wendel, M. Babarinow, T. Horl, S. Kolmogorov, S. Gobel, and R. Steinme, *Woodment: Web-based collaborative multiplayer serious game*, Transactions on Edutainment **IV** (2010), no. LNCS 6250, 68–78.

[41] J. Waldo, *Scaling in games and virtual worlds.*, Communications ACM **51** (2008), no. 8, 38–44.

[42] WC3, *Soap version 1.2*, http://www.w3.org/TR/soap12, 2007, Accessed: 2014 Feb 22.

APPENDIX A. TESTING

The Crushinator [1, 2] is a tool developed by WoWiWe Instruction Co. in collaboration with NDSU to verify the correctness of and load test a goal-based multiplayer simulation. A model of the goal system under investigation is written in the Unified Modeling Language (UML). This model is used by the Crushinator to construct multiple paths through the goal system, a form of Model-Based Exploratory Testing (MBET). These paths are then tested and analyzed for errors, alternate outcomes and other anomalies.

To perform load testing, a number of valid paths through the goal system are generated. A large number of automated players are created and assigned one of these paths to follow. This functionality was most helpful in choosing among alternative methods to display scenes, design simulations, and structure the goal system itself.

A.1 References

[1] C.J. Schaefer, H. Do, and B.M. Slator, *Crushinator: A framework towards game-independent testing*, Proceedings of 28th IEEE/ACM International Conference on Automated Software Engineering (Palo Alto, California), 2013.

[2] C.J. Schaefer, *Model-based exploratory testing: A controlled experiment*, Master's thesis, NDSU, Fargo, North Dakota, 2013.

APPENDIX B. VIRTUAL CELL API

Legend The following API definitions use example values and are in the form:

```
{
    client request
}
{
    server response
}
```

B.1 JavaMOO Namespace

Create Creates a new player account.

```
{
    "type": "Create",
    "username": "name",
    "password": "8jdsa8t43h",
    "hint": "my password hint",
    "email": "nobody@example.com"
}
{
    "type": "CreateResponse",
    "success": true,
    "result": true
}
```

Login Verifies the given player's password is correct, and returns a token for use with the remainder of the API events. This event places the player's avatar into the virtual environment, allowing other players to interact with the logged in player.

```
{
    "type": "Login",
    "username": "name",
    "password": "8jdsa8t43h"
}
{
    "type": "LoginResponse",
    "success": true,
    "result": true,
    "token": "92384nadfsa9b;4a8y243h",
    "hostname": "www.example.com",
    "port": 5001
}
```

Authenticate Verifies that the given player's password is correct, and returns a token for use with the remainder of the API events. This event is a meant to be used by the web interface, and does not place the player's avatar in any public rooms.

```
{
    "type": "Authenticate",
    "username": "name",
    "password": "8jdsa8t43h"
}
{
```

```
    "type": "AuthenticateResponse",
    "success": true,
    "result": true,
    "token": "92384nadfsa9b;4a8y243h"
}
```

Disconnect Removes the player's avatar from the virtual environment, then invalidates their token.

```
{
    "type": "Disconnect",
    "token": "92384nadfsa9b;4a8y243h"
}
{
    "type": "DisconnectResponse",
    "success": true,
    "result": true
}
```

GetExitDestination Queries an exit object URI for the room it leads to. This event is used to keep players out of areas they do not have access to, and will return `"result": false` if the player does not have access. Otherwise, the URI of the destination room is returned. This event may also be used to shard a room, should a room become too full (this has not yet been implemented).

```
{
    "type": "GetExitDestination",
    "id": "1~javamoo.domain.Exit~main",
```

```
    "token": "8jdsa8t43h"
}
{
    "type": "GetExitDestinationResponse",
    "success": true,
    "result": true,
    "name: "Needle",
    "id": "23~aux"
}
```

ChangeRoom Move the player's avatar to the room pointed to by the URI.

```
{
    "type": "ChangeRoom",
    "id": "23~aux",
    "token": "8jdsa8t43h"
}
{
    "type": "ChangeRoomResponse",
    "success": true,
    "result": true
}
```

GetRoomContents Retrieves a list of the objects in the current room. The client has a scene definition for each room, and is in charge of ignoring extra server-listed objects and making extra client prefabs (those without server URIs to match [Figure 8] them with) invisible. The client then sorts the URIs alphabetically, and the client prefabs by "closest to origin", and merges the two lists.

```
{
    "type": "GetRoomContents",
    "token": "92384nadfsa9b;4a8y243h"
}
{
    "type": "GetRoomContentsResponse",
    "success": true,
    "room": {
        "name": "Lobby",
        "id": "1~main"
    },
    "contents": [
        {
            "name": "Nucleus",
            "description": "<p>A healthy nucleus.</p>",
            "category": [
                "organelle"
            ],
            "prefab": "/Assets/Organelle/HealthyNucleus",
            "id": "3~aux",
            "created": 1342557888,
            "expires": 1344000000
        }
    ]
}
```

SendChat Sends the textual message to the named channel.

```
{
    "type": "SendChat",
    "token": "92384nadfsa9b;4a8y243h",
    "message": "Hello World.",
    "channel": "channel"
}
{
    "type": "SendChatResponse",
    "success": true,
    "result": true
}
```

GetChat Retrieves the messages from named channel over a given time period. `fromTime` and `timestamp` are represented as seconds since the epoch.

```
{
    "type": "GetChat",
    "token": "92384nadfsa9b;4a8y243h",
    "fromTime": long,
    "channel": "channel"
}
{
    "type": "GetChatResponse",
    "success": true,
    "messages": [
        {
            "username": "foo",
```

```
            "timestamp": long,
            "message": "hello bar."
        },
        {
            "username": "bar",
            "timestamp": long,
            "message": "hello foo."
        }
    ]
}
```

GetGoals Returns a list of goals that the player needs to accomplish to advance the game state.

```
{
    "type": "GetGoals",
    "token": "92384nadfsa9b"
}
{
    "type": "GetGoalsResponse",
    "success": true,
    "goals": [
        {
            "module": "TUTORIAL",
            "name": "Sample Goal",
            "description": "<p>This is a sample goal.</p>",
            "id": "89~javamoo.domain.goal~maindb",
            "tracked": true,
```

```
            "created": 31231456412,
            "tasks": [
                {
                    "name": "<p>This is a task name!</p>",
                    "description": "<p>This is a task!</p>",
                    "complete": false,
                    "completionTime": -1,
                    "id": "91~javamoo.domain.goal~maindb"
                },
                {
                    "name": "<p>This is a task name!</p>",
                    "description": "<p>This is the second task.</p>",
                    "complete": true,
                    "completionTime": 31231457852,
                    "id": "95~javamoo.domain.goal~maindb"
                }
            ]
        }
    ]
}
```


GetEncyclopediaTopics Returns a list of topics in the encyclopedia.


```
{
    "type": "GetEncyclopediaTopics",
    "token": "92384nadfsa9b;4a8y243h"
}
{
```

```
    "type": "GetEncyclopediaTopicsResponse",
    "success": true,
    "topics": [
        "topic1",
        "topic2"
    ]
}
```

GetEncyclopediaEntry Returns the HTML entry associated with the given topic retrieved from `GetEncyclopediaTopics`.

```
{
    "type": "GetEncyclopediaEntry",
    "topic": "nucleus",
    "token": "92384nadfsa9b;4a8y243h"
}
{
    "type": "GetEncyclopediaEntryResponse",
    "success": true,
    "entry": "<p>The <b>Nucleus</b> is ...</p>"
}
```

GetInventory Returns a list of objects in the player's inventory.

```
{
    "type": "GetInventory",
    "token": "92384nadfsa9b;4a8y243h"
}
```

```
{
    "type": "GetInventoryResponse",
    "success": true,
    "items": [
        {
            "item": {
                "name": "Pickaxe",
                "description": "<p><b>Pulverize</b> stone.</p>",
                "category": [
                    "item",
                    "tool"
                ],
                "prefab": "/Assets/Items/Tools/Pickaxe",
                "id": "123456~main",
                "created": 141451326,
                "expires": 0
            },
            "amount": 1
        }
    ]
}
```


GetStoreInventory Returns a list of objects in the given URI-referenced object's inventory.


```
{
    "type": "GetStoreInventory",
    "token": "92384nadfsa9b;4a8y243h",
```

```
    "id": "1~aux"
}
{
    "type": "GetStoreInventoryResponse",
    "success": true,
    "items": [
        {
            "item": {
                "name": "Pickaxe",
                "description": "<p><b>Pulverize</b> stone.</p>",
                "category": [
                    "item",
                    "tool"
                ],
                "prefab": "/Assets/Items/Tools/Pickaxe",
                "id": "123456~main",
                "created": 141451326,
                "expires": 0
            },
            "amount": 1,
            "price": 500
        }
    ]
}
```


AddItem Add an object, referenced by URI, to the player's inventory. The response contains the amount of the object now in the player's inventory.

```
{
    "type": "AddItem",
    "token": "92384nadfsa9b;4a8y243h",
    "id": "654321~main",
    "amount": 4
}
{
    "type": "AddItemResponse",
    "success": true,
    "result": true,
    "item": {
        "name": "Shovel",
        "description": "<p><b>Pulverize</b> dirt.</p>",
        "category": [
            "item",
            "tool"
        ],
        "prefab": "/Assets/Items/Shovel",
        "id": "654321~main",
        "created": 141451326,
        "expires": 0
    },
    "amount": 1
}
```

RemoveItem Remove the referenced object from the player's inventory. The response contains the amount of the object now in the player's inventory.

```
{
    "type": "RemoveItem",
    "token": "92384nadfsa9b;4a8y243h",
    "id": "654321~main",
    "amount": 1
}
{
    "type": "RemoveItemResponse",
    "success": true,
    "result": true,
    "item": {
        "name": "Shovel",
        "description": "<p><b>Pulverize</b> dirt.</p>",
        "category": [
            "item",
            "tool"
        ],
        "prefab": "/Assets/Items/Shovel",
        "id": "654321~main",
        "created": 141451326,
        "expires": 0
    },
    "amount": 1
}
```


UseItem Informs the target object that an object is being "used" on it. In order for the simulation interaction algorithm [Figure 9] to work as expected, `item`

should not be set to an object in the player's inventory. Send a `RemoveItem` (or `FireItem`) and use the URI reported in the `RemoveItemResponse`. This is due to internal containers ("stacks") that contain multiple objects, `RemoveItem` will peel one object off the stack for use.

```
{
    "type": "UseItem",
    "token": "92384nadfsa9b;4a8y243h",
    "item": "1546~main",
    "target": "123456~main"
}
{
    "type": "UseItemResponse",
    "success": true,
    "result": true
}
```

TradeItem In order to purchase objects from "stores", the player must request an item and offer what they want to trade in response. In order to see the "worth" of objects, use the `GetStoreInventory` event. The response contains the amount of the object now in the player's inventory.

```
{
    "type": "TradeItem",
    "token": "92384nadfsa9b;4a8y243h",
    "offering": {
        "id": "123361~main",
        "amount": 4
```

```
    },
    "asking": {
        "id": "654321~main",
        "amount": 1
    },
    "store": "45784155~main"
}
{
    "type": "TradeItemResponse",
    "success": true,
    "result": true,
    "items": [
        {
            "item": {
                "name": "Shovel",
                "description": "<p><b>Pulverize</b> dirt.</p>",
                "category": [
                    "item",
                    "tool"
                ],
                "prefab": "/Assets/Items/Shovel",
                "id": "654321~main",
                "created": 141451326,
                "expires": 0
            },
            "amount": 1
```

```
        },
        {
            "item": {
                "name": "ATP",
                "description": "<p>Energy Source</p>",
                "category": [
                    "item",
                    "tool"
                ],
                "prefab": "/Assets/Items/Atp",
                "id": "123361~main",
                "created": 141451326,
                "expires": 0
            },
            "amount": 6
        }
    ]
}
```

B.2 Virtual Cell Namespace

SetOrientation Sets the orientation (as a quaternion) of the player's avatar.

```
{
    "type": "SetOrientation",
    "x": 0.5,
    "y": 0.5,
    "z": 0.5,
```

```
    "w": 0.5,
    "token": "8jdsa8t43h"
}
{
    "type": "SetOrientationResponse",
    "success": true,
    "result": true
}
```

SetPosition Sets the 3D position of the player's avatar.

```
{
    "type": "SetPosition",
    "x": 0.0,
    "y": 0.0,
    "z": 0.0,
    "token": "8jdsa8t43h"
}
{
    "type": "SetPositionResponse",
    "success": true,
    "result": true
}
```

GetSceneState Returns a list of objects in the current room that have the internal "visible" property set to true. This event is used to track rapidly updating objects, such as players and torpedos, and synchronize them across all clients.

Also returned are the `callbacks`, the primary method the server uses to hint

to the client that the virtual environment has updated and which API call should be made to read the updated environment state. This event serves as the update-loop for the client.


```
{
    "type": "GetSceneState",
    "token": "92384nadfsa9b;4a8y243h"
}
{
    "type": "GetSceneStateResponse",
    "success": true,
    "scene": {
        "id": "1546~main",
        "name": "lobby",
        "health": 80.0
    },
    "callback": [
        "GetInventory",
        "GetChat",
        "GetGoals"
    ],
    "objects": [
        {
            "name": "457_120006000",
            "description": "<p>Regular torpedo</p>",
            "category": [
                "torpedo"
```

```
        ],
        "prefab": "/Assets/Item/Torpedo",
        "id": "4522~main",
        "created": 120006000,
        "expires": 120008000,
        "position": {
            "x": 0.1,
            "y": 0.1,
            "z": 0.1
        },
        "orientation": {
            "x": 0.1,
            "y": 0.1,
            "z": 0.1,
            "w": 0.1
        }
    }
],
"players": [
    {
        "name": "foo",
        "description": "<p><b>foo</b>, novice hydronaut</p>",
        "category": [
            "player"
        ],
        "prefab": "/Assets/Subs/Yellow",
```

```
            "id": "457~main",
            "created": 120006000,
            "expires": 0,
            "position": {
                "x": 0.1,
                "y": 0.1,
                "z": 0.1
            },
            "orientation": {
                "x": 0.1,
                "y": 0.1,
                "z": 0.1,
                "w": 0.1
            }"flair": [
                "/Assets/Flair/ID",
                "/Assets/Flair/ETC"
            ],
            "color": {
                "r": 0.4,
                "g": 0.5,
                "b": 0.1,
                "a": 1.0
            }
        }
    ]
}
```

GetLogBook Returns a list of historic events that have occurred since the given time.

```
{
    "type": "GetLogbook",
    "fromTime": 1320270201,
    "token": "92384nadfsa9b;4a8y243h"
}
{
    "type": "GetLogbookResponse",
    "success": true,
    "logs": [
        {
            "timestamp": 1320270205,
            "heading": "TutorialTask",
            "entry": "Completed",
            "success": true
        },
        {
            "timestamp": 1320270205,
            "heading": "TutorialGoal",
            "entry": "Completed",
            "success": true
        }
    ]
}
```

GetIncidentReportQuestions Retrieves a list of questions from the referenced "report" for the player to answer.

```
{  "type": "GetIncidentReportQuestions",
   "token": "8jdsa8t43h",
   "id": "42~incident.report.class",
}
{
  "type": "GetIncidentReportQuestionsResponse",
  "success": true,
  "result": true
  "questions": [
    {
      "id":"3072~report.question.class",
      "text":"is pluto a planet?",
      "answers": [
        {
          "id":"3073~report.answer.class",
          "text": "yes"
        },
        {
          "id":"3076~report.answer.class",
          "text": "maybe yes"
        }
      ]
    }
  ]
```

SubmitIncidentReport Submits the given "report" via its URI and provides player-supplied answers to its questions.

```
{
    "type": "SubmitIncidentReport",
    "token": "8jdsa8t43h",
    "id": "42~incident.report.class",
    "answer": {
        "question1": "answer1",
        "question2": "answer2",
        "questionN": "answerN"
    }
}
{
    "type": "SubmitIncidentReportResponse",
    "success": true,
    "result": true
}
```

ActivateRobot This event prompts an agent to give queue its dialogue for the player to read.

```
{
    "type": "ActivateRobot",
    "token": "8jdsa8t43h",
    "id": "42~robot.class"
}
{
```

```
    "type": "ActivateRobotResponse",
    "success": true,
    "result": true
}
```

ViewHealthyETC This event is a Remote Procedure Call that informs the server that the client has seen the ETC animation cycle enough times and/or the client has displayed the ETC movie to the player.

```
{
    "type": "ViewHealthyETC",
    "token": "92384nadfsa9b;4a8y243h"
}
{
    "type": "ViewHealthyETCResponse",
    "success": true,
    "result": true
}
```

FireItem Removes the referenced object from the players inventory, sets the removed objects orientation to the player's, and sets it's "visible" property to true so that other clients can independantly animate its trajectory. The response contains the amount of the object now in the player's inventory.

```
{
    "type": "FireItem",
    "id": "123456~main",
    "token": "8jdsa8t43h",
```

```
    "amount": 1,
    "position": {
        "x": 0.1,
        "y": 0.1,
        "z": 0.1
    },
    "orientation": {
        "x": 0.1,
        "y": 0.1,
        "z": 0.1,
        "w": 0.1
    }
}
{
    "type": "FireItemResponse",
    "success": true,
    "result": true,
    "item": {
        "name": "Torpedo",
        "description": "<p><b>BOOM</b>!</p>",
        "category": [
            "item"
        ],
        "prefab": "/Assets/Items/Torpedo",
        "id": "654321~main",
        "created": 141451326,
```

```
        "expires": 0
    },
    "amount": 99
}
```

ViewHealthyPhotosynthesis This event is a Remote Procedure Call that informs the server that the client has seen the photosynthesis animation cycle enough times and/or the client has displayed the photosynthesis movie to the player.

```
{
    "type": "ViewHealthyPhotosynthesis",
    "token": "92384nadfsa9b;4a8y243h"
}
{
    "type": "ViewHealthyPhotosynthesisResponse",
    "success": true,
    "result": true
}
```

ReplenishATP The idea was kicked around to have the player's avatar slow down after consuming "ATP", and to give them more after so many minutes, thus promoting small and numerous play sessions. The response contains the amount of the object now in the player's inventory.

```
{
    "type": "ReplenishATP",
    "token": "8jdsa8t43h"
}
```

```
{
    "type": "ReplenishATPResponse",
    "success": true,
    "result": true,
    "item": {
        "name": "ATP",
        "description": "<p>Energize!</p>",
        "category": [
            "item"
        ],
        "prefab": "common/Prefabs/ATP",
        "id": "654321~main",
        "created": 141451326,
        "expires": 0
    },
    "amount": 25,
    "added": 5
}
```

GetAvailableColors Returns a list of avatar colorings that can be applied.

```
{
    "type": "GetAvailableColors",
    "token": "92384nadfsa9b4a8y243h"
}
{
    "type": "GetAvailableColorsResponse",
    "colors": [
```

```
        {
            "r": 0.4,
            "g": 0.5,
            "b": 0.1,
            "a": 1.0
        },
        {
            "r": 0.7,
            "g": 0.0,
            "b": 0.1,
            "a": 1.0
        }
    ],
    "success": true
}
```

SetPlayerColor Sets the player's avatar to the given color.

```
{
    "type": "SetPlayerColor",
    "token": "92384nadfsa9b4a8y243h",
    "r": 0.4,
    "g": 0.5,
    "b": 0.1,
    "a": 1.0
}
{
    "type": "SetPlayerColorResponse",
```

```
    "success": true,
    "result": true
}
```

GetIncidentReports Returns a summary of the answered "reports".

```
{
    "type": "GetIncidentReports",
    "token": "92384nadfsa9b4a8y243h"
}
{
    "type": "GetIncidentReportsResponse",
    "reports": [
        {
            "id": "234~vcell.domain.tool.report.IncidentReport~maindb",
            "score": 0.5
        },
        {
            "id": "238~vcell.domain.tool.report.IncidentReport~maindb",
            "score": 1.0
        }
    ],
    "success": true
    "success": true
}
```

GetDialogue Removes a single dialogue from the player's dialogue queue, and returns it to the client for display to the player.

```
{
    "type": "GetDialogue",
    "token": "92384nadfsa9b;4a8y243h"
}
{
    "type": "GetDialogueResponse",
    "skin": "(Module)/Textures/Dialogue/dialogue_skin",
    "start": 0,
    "defaultImg": "(Module)/Textures/Dialogue/default_img",
    "defaultAudio": "(Module)/Audio/Dialogue/default_clip",
    "dialogue": [
        {
            "name":"Name",
            "text":"Text",
            "img":"(Module)/Textures/Dialogue/SubIntro/00_image",
            "choices":[
                {"text":"First choice", "next":0, "editor":False},
                {"text":"Second choice", "next":1, "editor":False}
            ],
            "choiceMode":"WHEEL",
            "links":[],
            "next":1,
            "editor":false,
            "pos":"BOTTOM",
            "passwords":[],
            "incorrect":0,
```

```
    "exit":{"text":"Exit", "next":0, "editor":False},
    "alignment":"LEFT",
    "mode":"TIMED",
    "narration":["/Dialogue/SubIntro/00_tutorial"],
    "script":"",
                "script_args":[],
    "duration":1.8
},
{
    "name":"Dr. Virchow",
    "text":"The Bixby-Klement.",
    "img":"/Textures/Dialogue/SubIntro/01_virchow",
    "choices":[],
    "choiceMode":"WHEEL",
    "links":[],
    "next":2,
    "editor":false,
    "pos":"BOTTOM",
    "passwords":[
        {"text":"Choice 1", "next":0, "editor":False},
        {"text":"Choice 2", "next":1, "editor":False}
    ],
    "incorrect":2,
    "exit":{"text":"Exit", "next":0, "editor":False},
    "alignment":"RIGHT",
    "mode":"TIMED",
```

```
        "narration":["/Dialogue/SubIntro/01_tutorial"],
        "script":"",
                    "script_args":[],
        "duration":1.3
    }
],
"cancelDialogue": {
    "type": "GetDialogueResponse",
    "skin": "(Module)/Textures/Dialogue/dialogue_skin",
    "start": 0,
    "defaultImg": "/Textures/Dialogue/default_img",
    "defaultAudio": "/Audio/Dialogue/default_clip",
    "dialogue": [
        {
            "name":"Dr. Moongrass",
            "text":"Oh, you think you know how to pilot the sub?",
            "img":"/Textures/Dialogue/SubIntro/cancel",
            "choices":[],
            "choiceMode":"WHEEL",
            "links":[],
            "next":0,
            "editor":false,
            "pos":"BOTTOM",
            "passwords":[],
            "incorrect":0,
            "exit":{"text":"Exit", "next":0, "editor":False},
```

```
                "alignment":"LEFT",
                "mode":"TIMED",
                "narration":["/Dialogue/SubIntro/cancel"],
                "script":"",
                                "script_args":[],
                "duration":5
            }
        ],
        "cancelDialogue": {}
    }
}
```