Lux Project

Philadelphia Game Lab

Jake Ailor

Introduction

Purpose

This Document is intended to provide an overview of the Lux Backend system design. It should provide reference to developers, as well as a reference to project advisors in order to help further the development process.

Scope

The Lux Backend system is a generic Multi-player game server-side system, intended for use with Massively Multi-player online games. The system is designed to be released open source for use by not only the Philadelphia Game Lab, but also other game development firms who are looking to develop online games without the resources that might be needed to develop a responsive server side process. In order to provide security for Multi-player games, an authentication system has been included in the design.

We believe that the Lux system is designed to be generic enough for use by almost any Multi-player game, while still being complete enough to exclude the need for further development on the server processes. Since the Lux system will be released open source, the code will be made available on GitHub for public access, however an installation kit (or make file) will be included with the final release.

The first project built on the Lux backend is a Massively Multi-player online game being developed in-house by the Philadelphia Game Lab. This will allow for close work in development of necessary features and improvements that may be excluded or overlooked in the initial release. Development on this MMO will begin once the server-side system has reached an appropriately useful stage.

Definitions and Acronyms

MMO – Massively Multi-player Online Game

BGT – Battle Ground Thread

HMBL – HashMap Based Location

EUID – End-User Identifier

System Architecture

The System is broken into several sections that interact with each other. The main functionality of the system is designed as a Message Passing Server that takes in Messages from client processes, finds the relevant recipients of the processes and forwards that message onto the recipient's socket. In order to determine which sockets the message is “relevant” for, we are developing our own data structure that functions as a 2D HashMap of the game Map. Sockets will be used for communication in order to speed the message passing system. In order to ensure that all clients are in sync with each-other, the client game process will pass all updates to objects/sprites in the game onto the server, and all messages will be passed back out.

A database of all objects will be kept to maintain a record of the current state of the game, and to allow new players to enter the game at it's current state. The handling and sending of object updates will be done on the client developer's end, however a reference guide will be provided to ease in this processes. Clients will be expected to send the “state” of all game objects, including the model that the game objects uses (store on the End-users local machine), the animation that the game model is currently in (and the time of the animation if necessary), and the location of the game object. Since objects will be updated on each frame, and only objects that the client has changed will need to be sent back to the server, we will minimize the traffic sent between frames without compromising game or introducing unnecessary lag into the game.

The game will provide a portion dedicated to Authorization, which will be an extension of the OAuth 2.0 protocol. Clients will include an OAuth 2.0 system that is external to our server system, and can either be of their own development or an external service such as Google or Facebook. To increase security, the OAuth, and in game Authorization are kept completely separate from the message passing server aspect, and from all other aspects of the server. In order to validate the End-user and verify that they are the user they claim to be, a static hashing method will be implemented that can hash a users credentials to check that they are the user in question.

Included with the system is a chat server, and banking system, both of which can be implemented either with the game server, or should function independently enough to be used as a stand alone system. The chat server will function as a chat server and miniature social network tool that can be integrated fully with a game, without restricting the game to any particular social or hierarchical structure. The banking system functions as a store of transactions for a particular user, or group, and acts independent from the rest of the system in order to prevent any conflict.

Architecture

The System is split into 10 separate modules that work as independently from one another as possible while still creating a full system. These modules are split into 3 major implementation types, HTTP servers, continual socket servers, and helper classes. The HTTP servers include an Authorization server, a Initialization Server, a Social Network server, and Bank server. These function on a HTTP protocol and will get in HTTP requests and return the output from a compiled program that is run when the request is received. The socket servers include a Battle Ground thread, and a chat server. These servers all listen on a socket, and receive messages before parsing them and sending a response to other sockets connected to the server. The helper classes include a static authorization class, a custom data structure for storing an end users position on the virtual landscape (HashMap Based Location), a Battle Ground Spawning thread, and a class designed to assign End-users to a given Battle Ground thread

(Find BGT Class). The functionality of these 3 modules varies greatly and will be discussed in further detail later.

HTTP Servers

The HTTP servers that are used within the game are portions of the game that do not need to function as quickly as other portions (eg: sockets), but still need to ensure reliable data transfer. The HTTP servers will work by receiving a formatted URL request to an HTTP server built into our functionality, which will be parsed for parameters. These parameters will determine the pre-compiled program that needs to be run, as well as the unique parameters to that program. As an example

https://localhost:3000/Auth?clientID=xxx&AccessToken=xxx

would run the Authorization server with the parameters supplied, and return the output from that program as the response to the request. The response will be a JSON formated string that can be parsed and interpreted in the client program.

“{

EUID: \“123\”,

AccessToken: \“xxx\”

}”

For the request processing we would like to use a fully functional HTTP server that can be adapted to return the output from our compiled program, opposed to adapting or building a smaller less capable HTTP server. For this purpose we will compile our programs as CGI compatible files, and configure our webserver to treat them as such. When an HTTP request comes in, a .htaccess file will reformat the URL, and a .config file will specify that the request is intended for a CGI compatible file.

Authorization Server

The Authorization Server is designed as an extension of the OAuth protocol, and acts as a next step in the Authorization process, without involving the End-User any further than an OAuth redirect for log-in. The only extra step that the client program must perform is a single HTTP request to the Authorization Server. After completing the OAuth process, the client program will forward the information returned (including the access token and unique client ID) to the Authorization server. The Authorization server will then validate the JWT formatted client ID as being a valid identifier and use this to access the players End-User ID from a key-value store (either Voldemort or Raik). The client ID will be validated using the static Authorization class.

Using the client ID returned from the OAuth process as the “key”, the key-value store will return the unique End-User ID that will be used in the game. The rest of the information returned by the OAuth will be updated in the key-value store, but will not be utilized at this point in the development, it is only stored for possible future features. The End-User ID is a unique identifier that is generated when a user first creates an account, and will need to be created within the Authorization server if the user is not already a registered player.

After the End-User ID is acquired from the key-value store, the Authorization server will call the static Authorization class again to create a unique in-game access-token. This access token will be a hash of the users End-User ID, a server secret, and a time bucket representing the current server time.

After creating this access-token, the HTTP server will return to the client the End-User ID, and the new access-token. The access-token is not stored in any database, but the client will need to use it to verify their identity on other servers.

Initialization Server

The Initialization server functions to get a single HTTP request in from a client, and respond with the Socket on which to connect, and the initial objects in the client game instance that need to be loaded. The request sent into the Initialization Server should include only the End-User ID and the access-token. The access token and End-User ID are then validated using the static Authorization Class. The End-User ID is then used to access a document database to retrieve the User's “Document”. This Document contains information on the player location, XP, and any other player specific information. This information is passed directly into the Find BGT Class, which uses the document information to determine which BGT the client should be connected to. It is highly likely that clients will wish to adapt this functionality specifically for their game.

After the call to the Find BGT Class, the Initialization server will take the returned information, including the BGT ID and the relevant “Bucket” numbers. The buckets will be described later in the HashMap Based Location Data structure. This information is then used to call the document database again, and find all the information that is relevant to the client at the start of their game play. The socket that the BGT is listening on and all the objects obtained from the document database are then passed back to the client as the response from the initial HTTP request. The client is then responsible for loading the Sprites saved on their local machine based on the information passed, and connecting to the socket specified by the response. Once the client finishes initializing the game session, they connect to the Battle Ground thread and begin game play.

Social Network Server

The Social Network server acts to receive requests from clients regarding joining and leaving groups, adding friends, and removing friends. A host of other features are included, but they all work in the same fashion, so they can be grouped together into a single functional description. The only difference between each call will be the database query that is used to make the change to the End-Users social network, or to retrieve information that will be returned.

The basis for both the social network and the chat server is the use of a graph database to establish and define the connection between users and groups. Each user and group will have an assigned Node, and the connection between the Nodes function as the “Friendship” or “Group Membership”. Each HTTP request that comes in will be formated to use a predefined query to the graph, which will substitute certain parameters to return the desired results, or edits.

The Social Network Server will get the HTTP request containing the action to be performed, the client ID, the client access-token, and the necessary parameters for the action. After using the static Authorization Class to determine that the client is sending an authentic request, the server will parse the necessary parameters into the desired query, and perform the query on the database. The response from the HTTP request will be the response from the database, which will be either a list of desired nodes/connections, a success message, or a failure message. Failures may result from attempting to make a query on a non-existent connection or node, but should not occur if the social network is properly used by the client program.

Bank Server

Like the Social Network Server, the bank server will function as a large HTTP API that can take in a certain defined set of changes, and act almost directly on the database from the given parameters after proper Authentication from the Static Authentication Class. In order to add a second layer of security, there may be some need for PIN code access or IP verification during a session.

The basic functionality will be receiving, transferring, and spending of in-game currency. Since the full functionality of the Bank Server, and the necessary parameters have not been defined, this section will be left unexplored.

Socket Servers

The Socket Servers are designed to function as message passing servers that receive a message on their own listening socket, perform minimal processing and pass the message onto a sending queue that sends messages on it's own sending socket. In order to keep the systems response time up (and the client lag down) the Socket servers preform as little work on each message as possible before sending them out. The main functionality within each of the receiving sockets is to find which sockets will need to receive each message, and forward the message onto a sending thread.

Battle Ground Thread

The Battle Ground Thread holds the main functionality for the server, which is receiving and updating the objects during the game play. The BGT will first open a socket connection to listen to, and then receive messages on the socket to be parsed into JSON objects. These objects will contain a header which contains the client that sent the message, and the location of that client. The first query to the HBML will function to update the location of the player who sent the object update, and remove the reference to the old location. It will also update the socket that the client is using- in case the client is forced to choose a new socket for any disconnection reason, without forcing the client to reconnect via the Initialization server.

The second call the the HMBL will be the location of the object itself, and the radius of which clients will need to be notified of this change. The radius portion will be static to the object for the most part, although client processes will be able to alter this. The HMBL call will return a linked list of the sockets relevant to this update. This object update, and the list of relevant sockets will then be piped off to a Send Updates thread that is unique to the BGT. The Send Updates Thread will function only to traverse the linked list of sockets and send the object update to each of the client sockets that the update is relevant to. After sending the update to all sockets, the Send Update Thread will pipe the message to a Database Writer thread that will update the object definition in the document database. This update, and the document database, will not be accessed at any point during the game play, but is only used by the initialization thread to prepare a new client who is preparing to enter the Battle Ground Thread.

Chat Server

The Chat Server functions along-side the Social Network Server, and works off the same graph database as the social network server. The functionality however is very different, and more similarly reflects that of the Battle Ground Thread's functionality. As a message comes into the Chat server on the socket it is listening on, it will parse the message into JSON, query the graph database for a list of relevant sockets, and send out the message to the list of sockets via it's own unique Send Updates

Thread. On each message that comes in, the header to the message will contain the Sending user, who will need to be authenticated using the Authentication class, and the End-User's node in the graph database will have to updated with the socket that the message came in on.

Helper Classes/Structures

The Helper Classes and Structures are the aspects of the server that do not directly communicate with the client process, but instead are accessed by specific portions of the server for specific purposes. The Authorization class is accessed by any process that is receiving a message or sending a message to a client. The HashMap Based Location is created as an instance inside the Battle Ground thread and is specific to that Battle Ground thread. The Battle Ground Thread Spawner is only concerned with data in the Find BGT Class, and ensures that the proper BGT and proper number of BGTs are currently online; it is the only Helper class that runs continually- and will be used as the main thread that launches all the other threads and ensures that they are still online. The Find BGT Class will share many elements with the HMBL class, and will have many of it's methods statically called from the HMBL class, but will run separate to the BGT as a way to track the BGTs and determine the proper BGT for a client to connect to.

Authorization Class

The Authorization class is designed to be a static class, accessible from any other server component, in order to authorize clients without communication between individual threads. The functions within the Authorization class are static and the class will be called from various points in the system. The methods include a method to create an access token from the End-User ID, the Time bucket, and the server secret. One to verify that the access token matches the End-User ID, which will rehash the token and compare it to the token passed in. A method to authenticate the JWT from the OAuth process, which will follow the standards laid out for Authenticating a JWT, which will return the unique identifier from the JWT.

Lastly, the Authorization Class will contain a method for refreshing a User token if it has expired, which will rehash the passed token using the previous time bucket and make sure that it matches, and then create a new token in the current time bucket and return that. This method will be called anytime that a access token fails, but will need to be called from outside of the class.

HashMap Based Location

The HashMap Based Location tool is the heart of the systems ability to quickly process which clients need to be updated of changes to an object, and quickly determining the location of an object relative to other objects on the virtual map. The actual HMBL stores only the clients and their respective sockets, which allows for quickly compiling a list of sockets that a message needs to be sent to. The HMBL tool is instanced within a BGT and is not a shared data structure, so there are no shared locks, mutual exclusion, or blocking- only the BGT accesses the HMBL.

The structure of the HMBL tool is predicated on the use of two HashTables. The first HashTable is used only to hold relationship between a client and a pointer to their socket in the second HashTable. This is done so that when a request to update the client's socket comes in, the HMBL can immediately find and unlink the previous socket. As a precaution, sockets are given an “expiration time” when they are logged into the second HashMap which will be checked against the current time when the socket is

accessed. If the socket has expired (without having been previously unlinked) it is assumed that the client connection was terminated, and thus that socket no longer needs to have messages sent to it. This leads to the HMBL immediately unlinking the socket and continuing with it's task.

The second HashTable is a linear array of buckets that can be mapped to using a hashing function that is dependent on the clients current x,y coordinates on the map. The bucket's virtual size is dependent on the needs of the game, but is sized sufficiently small that a limited number of sockets are mapped to any given bucket. The HMBL's ability to return “relevant” sockets based on the radius of the object is dependent on the linear array's ability to have each element accessed in constant time given the index. If a socket is in a given bucket, and the radius is 1 level out, then each of the buckets around it can be found using a formula. This gives access to the buckets at constant time, assuming the buckets can be mapped to in constant time, and the formula to find the surrounding buckets is constant time. The only growth is for elements with a larger radius, there will be a large growth between the amount of buckets that need to be accessed. Since within each bucket the sockets will be stored as a (presumably doubly) linked list, it is still possible to link them together in constant time by not traversing the list, but instead just linking them to the end of the previous bucket's list.

When a socket is updated, the HMBL returns a list of relevant buckets that has changed in comparison to their previous location (eg: {0,1,2,3,4,5} vs {2,3,4,5,6,7} returns {6,7}). This result, as well as the socket, are piped into an Update buckets thread. This thread will function to query the database and send the relevant information to the client that has moved to a new thread. Due to bucket sizes, this will mean that information just off the edge of the map will be kept continually up to date as the client/player moves across the map.

Battle Ground Spawner Thread

The Battle Ground Spawner Thread functions just to maintain the BGT's and grow the map, or the number of arenas currently in use. As the number of Battle Ground Threads needed grows, the Battle Ground Spawner Thread will spawn a new BGT. This process is done by first duplicating a document collection that reflect the starting state of the battle ground thread being spawned (loading map elements, setting the spawn points, ect) and linking that to the Battle Ground thread that is being spawned, as well as to the Find BGT class.

If a BGT should crash, or go down, then the Spawner thread would simply request the database collection of that thread, and create a new BGT that is linked to the existing document collection. Since BGTs will be launched within the Battle Ground Spawner Thread, it will be able to see if any BGT has crashed, as well as kill any BGT that is no longer necessary.

Find BGT Class

The Find BGT class maintains an active list of the BGTs that are being played on, and contains a custom hashing function dependent on the game type to find which thread a client should be linked to. Once a client connects to the initialization server, the Find BGT Class will be able to use static methods, and simple definitions of each BGT to calculate which BGT the client is best matched with. Depending on game type, this could be based on the location of the client, the XP of the client, or the crowding of any given BGT. Each type of mapping will be implemented, and client Developers will simply need to specify which option they wish to use when the process is launched.

The Find BGT Class will access many of the statically defined methods of the HMBL to locate

the position on the map that the client is currently trying to access, and be able to find all the relevant buckets that will need to be queried from the document database. This separation allows the BGT to remain solely accessible from the BGT, while not completely isolating the information that would be needed to connect a client to the BGT.

Design Rationale

The most important considerations for the project came from the desire to increase security, implement as many generic features as possible, all while maintaining a high degree of speed and stability to the system. The object updates, and chat on the client end are the element most focused on speed, while not having as much concern about the occasional loss of an update. On the other side, most other aspects of the project (authorization, initialization) are not as concerned about speed, but very concerned with reliability and security. Due to this split, two separate design techniques were implemented that gave each portion the focus they needed.

The first technique, focused on a BASE style design, of being more or less unconcerned with needing every client to be precisely correct on every object at all times, but more concerned with every client being up to date on most objects most of the time. This was done in an attempt to reduce lag on the client end. If a few objects are not perfectly in sync, the clients view of the game will change only minimally due to the high frame rate of the game, and the fact that an object currently being updated, will likely be updated again very soon (probably the next frame), and the system will converge on correctness. This methodology contributed significantly to the use of NoSQL databases, as did the adaptability and the ease of finding a NoSQL solution that could be used to directly solve the individual problems faced in data storage by each module.

For the same reason, we attempted to reduce the number of actions that needed to be performed on any packet that needed to be processed quickly. By attempting to keep as much of the time complexity on each message consistent and low, the system is able to quickly respond to incoming messages. This introduced the problem of putting more stress on the client machine, and giving the client machine more responsibility to handle game logic. This gives clients more control over the game, but also opens the games up to unauthorized hacking, and possible discrepancies between client's game play.

The second technique was not as focused on the need for speed, but instead could fall into the category of “Fast enough”. Fast enough means that we were willing to sacrifice a few hundred milliseconds to make the API much simpler, reduce the number of socket connections that needed to be open, and use existing internet protocols to increase security and ease of use. For modules such as Authorization and initialization, this method was chosen because it was deemed more important for the entire message to be sent and received then it was for the message to be sent quickly.

Overall the design mainly focused on creating a reusable system that was not too heavily linked to any single game style or game play. The use of smaller processing of actual game objects helps to move our game into that realm. By focusing on game aspects that are present in a large number of games, including authorization, banking, client updates, and chat, we are able to appeal to a large audience of front-end game developers.

Component Design

Developer 1

Initialization Server

```
Parse parameters into
     Access-Token
     End-User ID
Query Document Store for End-User Document
Parse Document into JSON
Pass JSON Document to Find BGT Class
     return BGT ID & int[] Bucket List
Query Document Store for BGT Document
Query Document Store for BGT Objects in relevant Buckets
Build JSON response and return to client
```

Battle Ground Spawner Thread

Section missing

Send Updates Thread

```
Open Database Writer Pipe
Spawn Database Writer Thread
LOOP:
     Read Message from Pipe
     Break Message into Socket List & Message
     Pipe Message to Database Writer Thread
     Iterate through Socket List & Send Message to Each
```

Developer 2

HashMap Based Location

Section missing

Find BGT Class

Section missing

Developer 3

Battle Ground Thread

```
Create new HMBL
Open Send Updates Pipe
Spawn Send Updates Thread
Open Socket
LOOP:
     Read message from socket
     Parse Message into JSON
     Read Message Header for Access-token/EUID
     Authenticate Access-Token // Authentication Class
     Read Message Header for User Location
     Query HMBL to update User Location
          Return linked list of relevant sockets
     Strip message Header
     Pipe message body and Sockets to Send Updates Thread
```

Developer 4

Social Network Server

```
Parse Parameters into Command and Args
Use Switch to choose Appropriate Command
Query Graph Database with Appropriate Command
Return the results from the graph Database to client
```

Chat Server

```
Open Send Updates Pipe
Spawn Send Updates Thread (w/ DatabaseWrite = false)
Open Socket
LOOP:
```

```
Read Message from socket
Parse Message into JSON
Read Message Header for Access-token/EUID
Authenticate Access-Token // Authentication Class
Read Message Header for recipient(s)
Query Graph Database to Update Sender Socket
Query Graph Database for recipients
     Return JSON of recipient Sockets
Pipe Message body and Sockets to Send Updates Thread
```

Developer 5

Authorization Server

```
Parse parameters into
     string Access-Token
     string Unique ID
     string Client API Key
     string Refresh Token
Authenticate Unique ID // Auth Class
Check KV store for Unique ID
     Return End-User ID
     OR create new End-User in KV store
Update Access-Token in KV store
Hash a new Access-Token // Auth Class
JSON response to HTTP request
```

Developer 6

Bank Server

```
Parse Parameters into Command and Args
Use Switch to choose Appropriate Command
Query HM Store with Appropriate Command
Return the results from the HM Store to client
```

Authorization Class

string createAccessToken(string EndUserID, string serverSecret)

Get Current Time bucket

Hash Parameters + time bucket using... ?

Return new Hash as access-token

string authenticateJWT(string JWT)

Parse to JSON

Authenticate JWT

Return Unique ID from JWT

int authenticateUserID(string AccessToken, string EUID, string SS)

Call CreateAccessToken with parameters

Compare new hash to AccessToken (string compare)

return the compare

String refreshUserToken(string AccessToken, string EUID, string SS)

Get Previous Time bucket

Hash Parameters + time bucket using... ?

Compare new hash to AccessToken (string compare)

If compare == true

    Call CreateAccessToken

    return new AccessToken

Client API Design

System Options

Authorization Server – URL Request

Request:

Auth

Parameters:

AccessToken – The access token is the same access token that is returned from the OAuth Server

IDToken – The ID token is the same ID Token in JWT format returned from the OAuth

RefreshToken- From the OAuth

ClientAPIKey- From the OAuth

Returns:

AccessToken – The access Token to be used within the game for Authentication

EndUserID – The Users Unique in game Identifier

Initialization Server – URL Request

Request:

Init

Parameters:

AccessToken – The access Token to be used within the game for Authentication

EndUserID – The Users Unique in game Identifier

Returns

BGT Document – the Document used to identify the BGT, including map model

ObjectDocuments – A JSON object containing all objects that need to be initialized on the map

Social Network Server – URL Request

Request:

Parameters:

Returns:

Bank Server – URL Request

Request:

Parameters:

Returns:

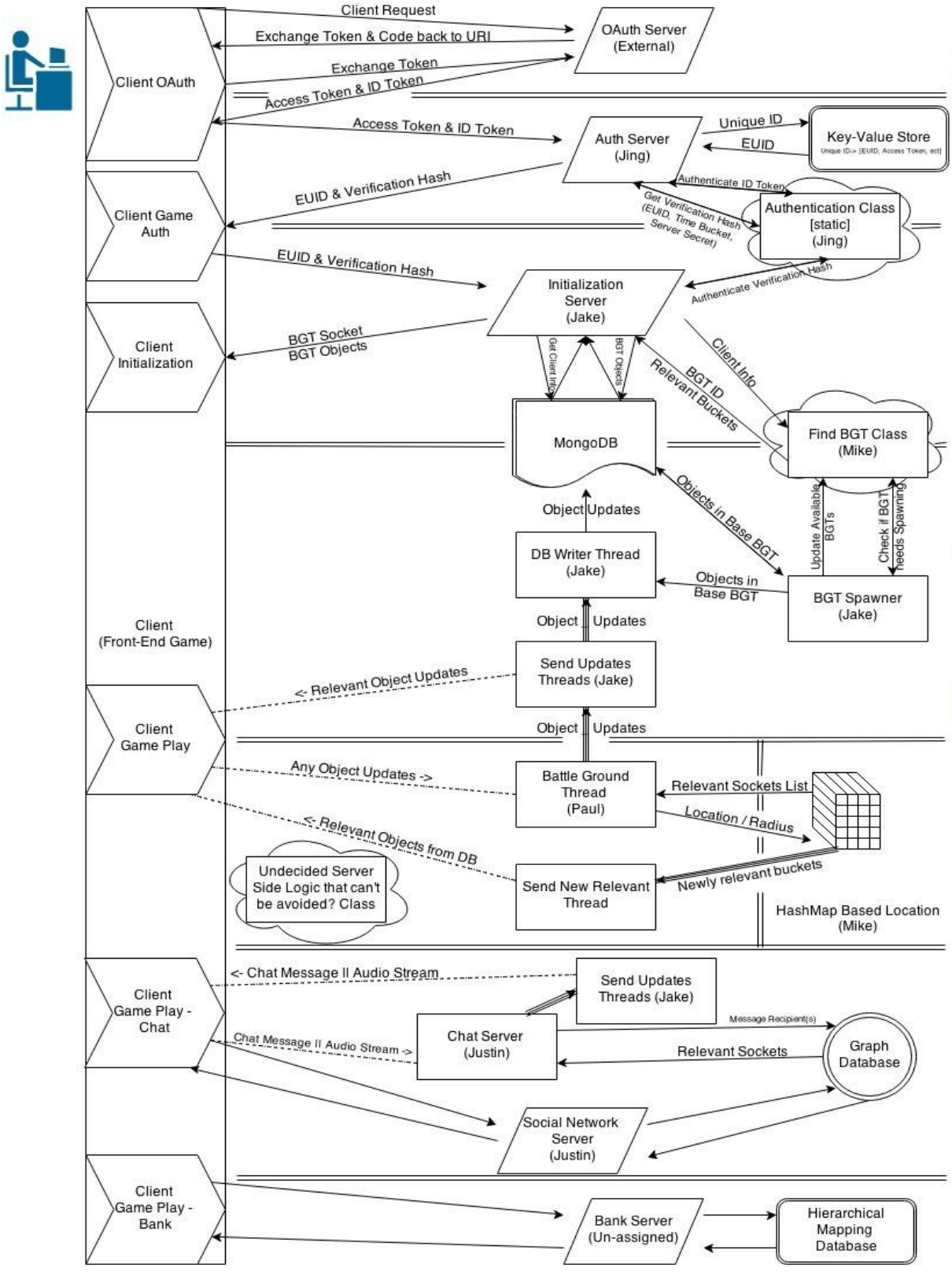
Chat Server – Socket Connection JSON

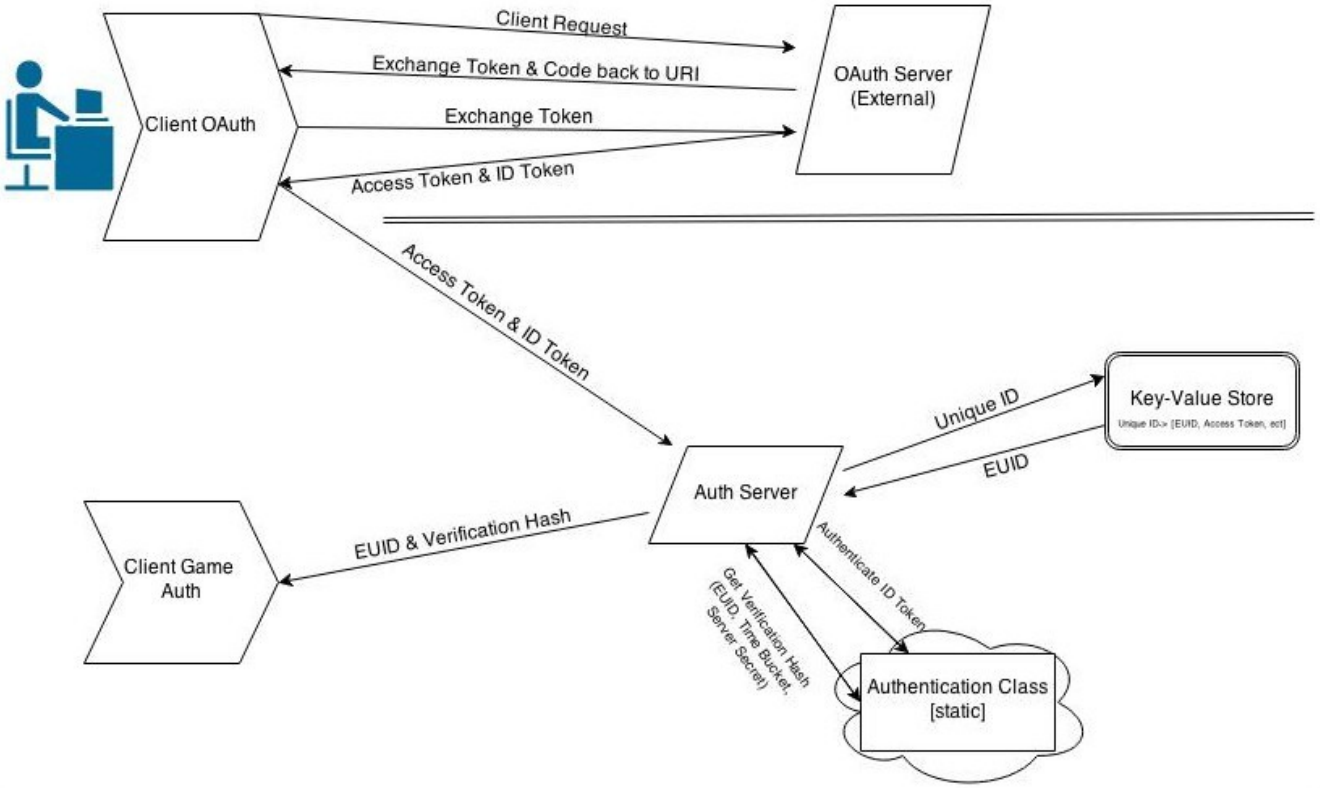
Needed Parameters:

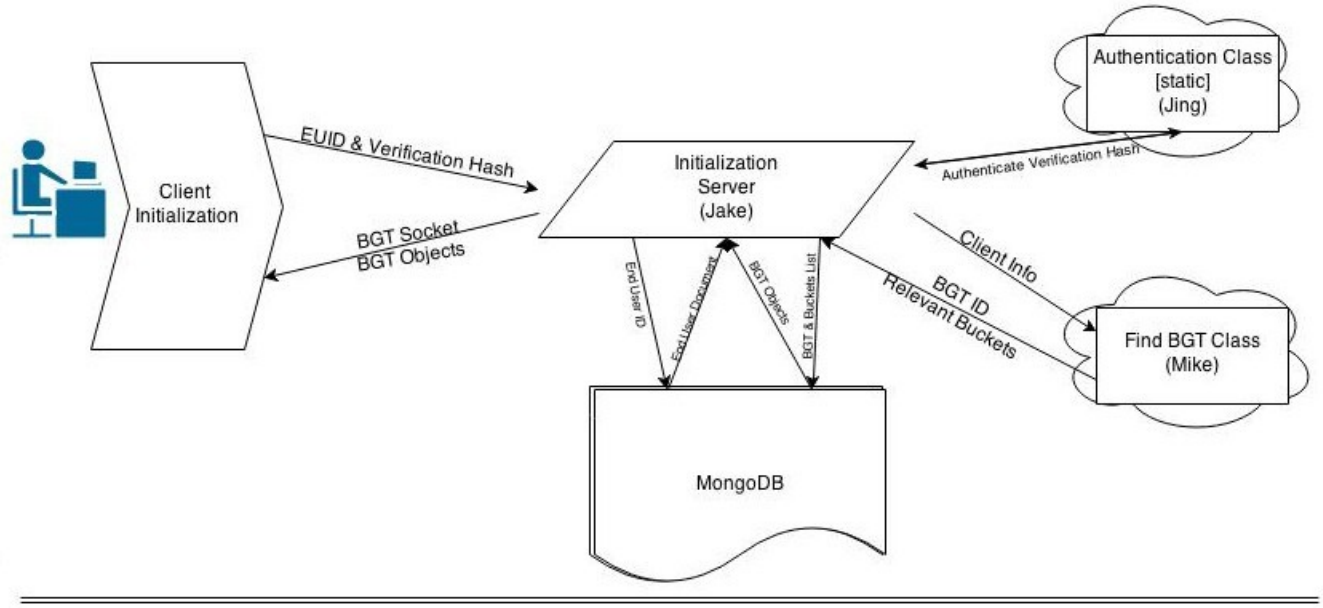
Battle Ground Thread – Socket Connection JSON

Needed Parameters:

System Design Images




Client Request
Exchange Token & Code back to URI
Exchange Token
Access Token & ID Token
Client OAuth
OAuth Server
(External)
Access Token & ID Token
Unique ID
EUID
Key-Value Store
Unique ID-> (EUID, Access Token, ect)
Auth Server
EUID & Verification Hash
Client Game
Auth
Authenticate ID Token
Get Verification Hash
(EUID, Time Bucket,
Server Secret)
Authentication Class
[static]


Authentication Class
[static]
(Jing)
Authenticate Verification Hash
Client
Initialization
EUID & Verification Hash
BGT Socket
BGT Objects
Initialization
Server
(Jake)
Client Info
BGT ID
Relevant Buckets
Find BGT Class
(Mike)
End User ID
End User Document
BGT Objects
BGT & Buckets List
MongoDB