**Proposed Alternative Matching Server (C#) for 4:33 Network Engine Project**

**Mik Minier**

**Proposal Version 1.0.0.6**

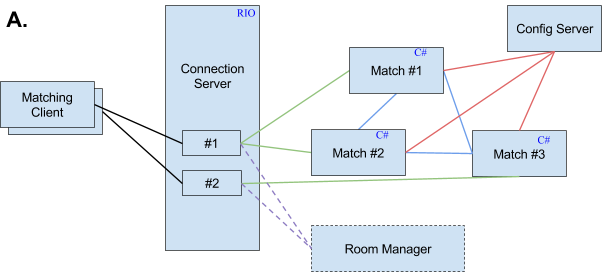
|  |  |  |  |
| --- | --- | --- | --- |
| Version | 날짜 | 변경 내역 | 이름 |
| 1.0.0.0 | 2016.12.22 | 초안 | 밐 |
| 1.0.0.1 | 2016.12.23 | +Data Structure/Synchronization Discussion | 밐 |
| 1.0.0.2 | 2016.12.27 | +Program Flow Diagram, +Index | 밐 |
| 1.0.0.3 | 2016.12.28 | +Coding specifications | 밐 |
| 1.0.0.4 | 2017.01.02 | +Logging methodology choice | 밐 |
| 1.0.0.5 | 2017.01.04 | +General Update | 밐 |
| 1.0.0.6 | 2017.01.05 | +Diagram Color Explanations | 밐 |
| 1.0.0.7 | 2017.01.09 | +Updated Program Flow | 밐 |

**Index:**

1. **Project Description**
2. **Matching Server Function Specifications**
   1. **Server Connectivity**
      1. Configuration Server
      2. Other Matching Servers
      3. Connection Server
   2. **Player Connectivity**
      1. Through client directly
      2. Through Matching Servers
      3. Through Connection Server
   3. **Match Making**
      1. Rules and iteration methods
   4. **Program Flow Diagram**
3. **Coding Specifications**

**I. Project Description**

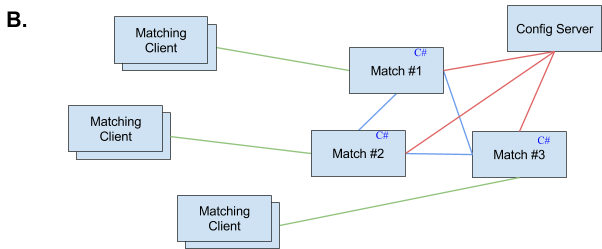
**Project Description:** Develop a matching server using C# to fit within either of the following models (A or B):



**Note:** Line Color represents a particular connection type (ie. CS to MS). Dotted lines mean optional.

Matching Data Flow

1. connection server에 room server와 matching server가 붙는다.
2. connection server는 client를 받는다.
3. client가 매칭 요청을 matching server로 보낸다.(CS를 통해)
4. matching server들은 client의 stat을 보고 비슷한 client와 매칭을 한다.
5. matching server가 room server에게 매칭 결과를 보낸다.
6. room server는 매칭 결과로 방을 만들고 matching server에 결과를 보낸다.
7. matching server는 client에게 매칭완료를 알린다.
8. client는 room server에게 방 참가를 요청.
9. room server는 client에게 game start 전송.
10. 방 참가 10~20 초 후 room server는 room 내의 client들에게 game end 메시지 전송.
11. 3으로 돌아감.



**Note:** Line Color represents a particular connection type (ie. MC to MS).

Matching Data Flow

1. matching server에 client들이 붙는다.
2. client가 매칭 요청을 matching server로 보낸다.
3. matching server들은 client의 stat을 보고 비슷한 client와 매칭을 한다.
4. matching server는 client에게 매칭완료를 알린다.
5. 2으로 돌아감.

**II. Matching Server Function Specifications**

**Sub-index:**

**1. Server Connectivity**

-> Configuration Server

-> Other Matching Servers

-> Connection Server

**2. Player Connectivity**

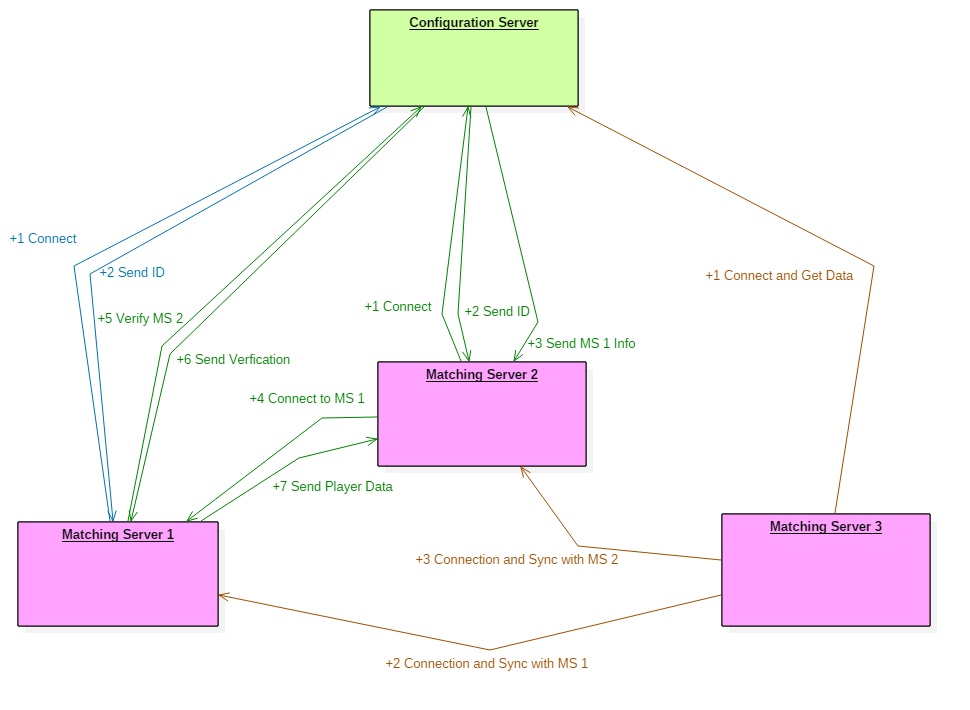
-> Through client directly

-> Through Matching Servers

-> Through Connection Server

**3. Match Making**

-> Rules and iteration methods



**Note:** The blue lines represent the connection process of the first MS, the green lines represent the connection process of the second MS, and the orange lines represent the connection process of the third MS with the configuration server (Green). The MS are colored pink.

**1. Server Connectivity**

-> *Configuration Server*

[**Purpose**] The ConfigServer stores MS IDs and IP addresses to control the MS network. It also dictates the addition of new MS in the system

[**Initialization**]

-1 The MS attempts to connect to the ConfigServer

[**FailCase**]

The MS retries connection to ConfigServer

-2 The MS sends a ID request to the ConfigServer

[**FailCase – socket error**]

The MS closes the socket and reattempts connection

-2 The ConfigServer verifies/registers the MS and sends the MS its ID

[**FailCase – socket error**]

The MS closes the socket retries connection with ConfigServer

-3 The MS opens an asynchronous process to receive messages from the ConfigServer

[**Synchronization**]

For each previously active MS:

-4 The ConfigServer sends info (ID, IP) about an MS-A to MS-B

[**FailCase – socket error**]

Continue initializing the MS without loading other MS information

-5 The MS-B connects with MS-A

[**FailCase – socket error**]

Continue initializing the MS without loading other MS information

-6 MS-A sends a verification request to the ConfigServer

[**FailCase – socket error**]

MS-A can’t verify MS-B so MS-A closes connection with MS-B

-7 The ConfigServer verifies MS-B with MS-A

[**FailCase – not verified**]

MS-A closes connection with MS-B, suspecting it to be fraudulent

[**FailCase – socket error**]

MS-A can’t verify MS-B so MS-A closes connection with MS-B

-8 MS-A registers MS-B in its WaitingRoom and calculates the initial latency

[**FailCase – out of memory**]

MS-A can’t support match sharing with MS-B so MS-A closes connection with MS-B

-9 MS-A sends its PlayerInfo to MS-B

[**FailCase – socket error**]

The MS (both sides) close the connection and clean-up relevant resources (WaitingRoom)

If all MS are registered, the MS initializes its processes

[**ConfigServerDown**]\*\*\*

If the ConfigServer goes down, while the game service is running, we should attempt to continue game service

Reconnection attempts will occur in the background

-> *Other Matching Servers*

[**MatchingServerDown**]

If the connection with a particular MS fails, we must remove that server’s players from the WaitingRoom

-1 Remove server listing and players, close connection with that MS

-> *Connection Server*

[**ConnectionServerDown**]

If the connection with the connection server fails, we can act as if all players on the server have disconnect

-1 Remove all local players from WaitingRoom

-2 Broadcast to all MS to remove all players from the MS at hand (special command)

-3 Attempt to reconnect to CS, while receiving updates about other MS players

**2. Player Connectivity**

-> *Through client directly*

[**Connection**]

-1 The client makes a connection with the MS

-2 The MS accepts the socket and awaits receive for a MatchingRequest command

[**FailCase – socket error**]

The MS closes the socket resources

-3 When a MatchingRequest command is received, the MS makes an ID for the player

-4 The MS calculates a metric for the player and submits the player to the WaitingPool with status UNMATCHED

[**FailCase – out of memory**]

-a Send ServerBusy message to player

-b Close socket

-5 The MS broadcasts the player ID/metric to all MS

[**Disconnection**]

-1 The MS removes the player from the WaitingPool

-2 The MS broadcasts to all MS about the removal

-> *Through Matching Servers*

[**Connection**]

-1 Another MS reports a new player with ID and metric

-2 The MS adds the new player data to the WaitingPool

[**FailCase – out of memory**]

Special case – send retry message back to another MS?

[**Disconnection**]

-1 The MS removes the player from the WaitingPool

-> *Through Connection Server*

[**Connection**]

-1 The MS receives a MatchingRequest command from a client through the connection server

-2 When a MatchingRequest command is received, the MS makes an ID for the player (using CS ID code)

-4 The MS calculates a metric for the player and submits the player to the WaitingPool with status UNMATCHED

[**FailCase – out of memory**]

Send ServerBusy message to player

-5 The MS broadcasts the player ID/metric to all MS

[**Disconnection - CS**]

-1 The MS removes the player from the WaitingPool

-2 The MS broadcasts to all MS about the removal

[**Disconnection - Timeout**]

-1 The MS removes the player from the WaitingPool

-2 The MS broadcasts to all MS about the removal

**3. Match Making**

-> *Rules and iteration methods*

[**Rule1**]

Each MS matches one of its own players with either one of its own players or another MS’s player

[**Rule2**]

A match can only be made between two MSs if their latency value is under a certain threshold

[**Rule3**]

The matching search algorithm prioritizes players who have been waiting longer

[**Rule4**]

With latency bounds, the matching search algorithm considers players across different MSs fairly

[**Rule5**]

A match is determined by the differential between two players’ metrics

If the difference is within a threshold, a match is made

[**MatchMakingAlgorithm**]

Each MS’s players (PlayerInfo) are stored within an OrderedDictionary (OD); the data structure choice being:

-> Rapid look-up of any player needed for removal after match making

-> Order to maintain a waiting time relationship for fairness in matching

[**Step1**]

Enumerators are retrieved from each latency-eligible server

[**Step2**]

The local MS’s OD is scanned to find the first player with status UNMATCHED

[**Step3**]

Starting with the selected player’s index+1, other players are evaluated

(Note: index+1 is to avoid some potential loop-like cross server conflicts from arising)

[**CaseUNMATCHED**]

-1 Calculate the metric difference between two players

-2 If within the threshold, set the match

-3 Else, compare to the best previous match

-4 If better, save the current spot, otherwise move on

[**!CaseUNMATCHED**]

Move to next iteration as described below

-> Iterate through all latency-eligible OD at the current index before increasing the index

-> During index increase, relax the threshold slightly

-> Repeat Step3 until a match is found or not

[**CaseMatchFound**]

Match is found!

[**CaseMatchNotFound**]

In this case, no match was found within threshold, so the best saved match is used as the match

[**CaseNotEnoughPlayers**]

Need to handle this special case when only up to 1 player is available per server

[**MatchConfirmationProcess**]

[**CaseMatchWithSameMS**]

-1 The status of the two players is changed to CONFIRMED\_WITH\_SERVER

-2 The MS broadcasts the status change to all MSs

NOTE: At this point, we would confirm first with the Room Manager before proceeding

-3 The MS sends result to the players and closes their connections

-4 The MS broadcasts the status change to all MSs

[**CaseMatchWithDifferentMS**]

-1 The status of the two players is changed to MATCHED\_BUT\_UNCONFIRMED

-2 The match request is made to the other MS

[**FailCase – connection with other MS dead**]

-The local player status is reverted to UNMATCHED

-The other player is removed

[**FailCase – match denied**]

-The players’ statuses are reverted to UNMATCHED

-3 The other server checks confirms the assignment and sets the match to CONFIRMED\_WITH\_SERVER

[**FailCase – connection with original MS dead**]

-Revert player status to UNMATCHED

-4 The status of the two players is changed to CONFIRMED\_WITH\_SERVER

-5 The MS broadcasts the status change to all MSs

NOTE: At this point, we would confirm first with the Room Manager before proceeding

-6 The MS sends the result to the other MS to send the matching success and info

[**FailCase – connection with other MS dead**]

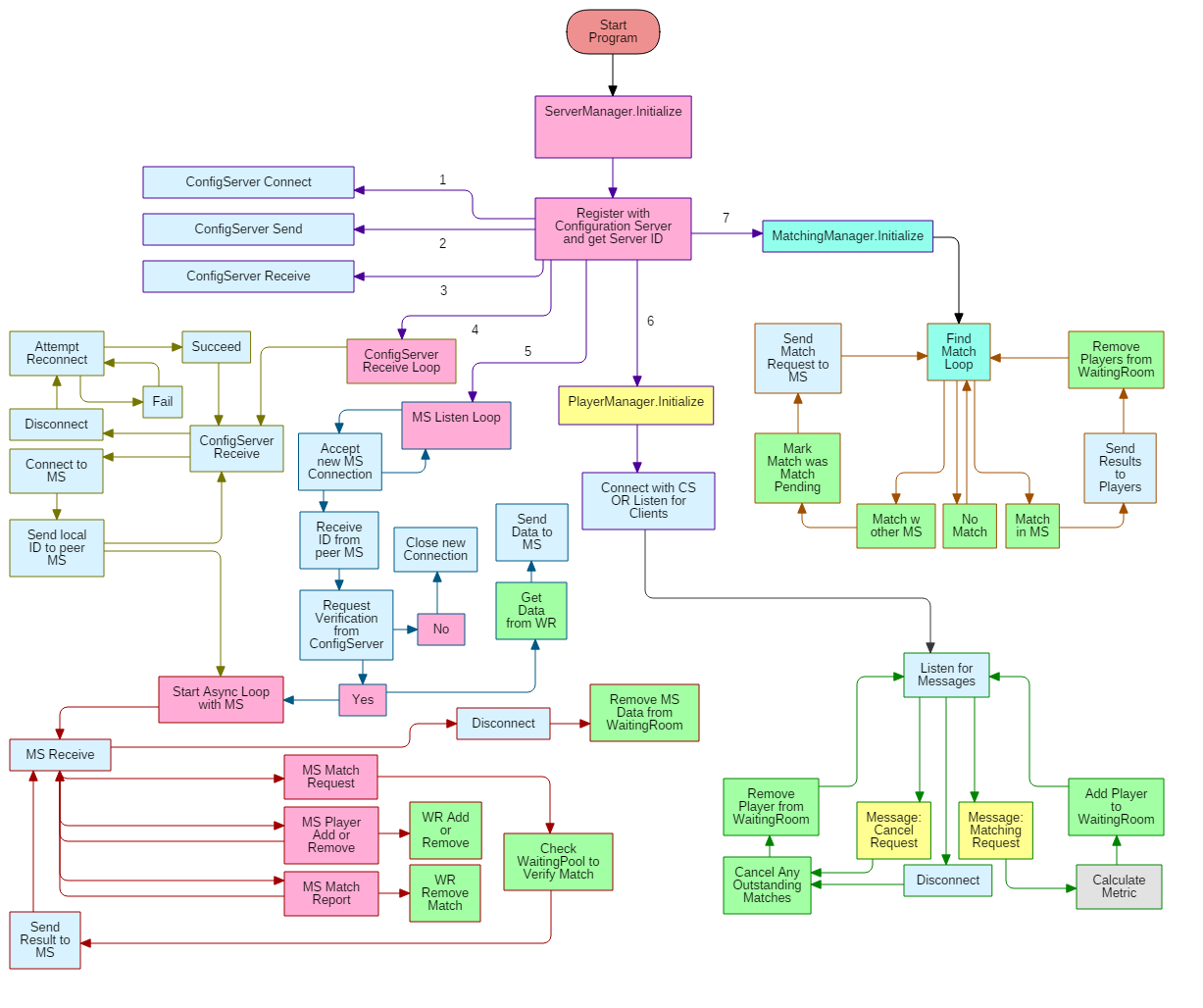
-The local player status is reverted to UNMATCHED

-The other player is removed

-7 The MS sends result to the local player

-8 The MS broadcasts the status change to all MSs

**Program Flow:** The program flow can be represented in the following diagram. The purple lines represent the program setup and initialization and the other colors represent async/await loops. Box colors represent which class runs the particular method or logic.



**Note:** Line colors represent a region of logical code flow (a loop) whereas box colors represent class ownership.

*Classes:* Pink = ServerManager; Blue = ConnectionManager; Green = WaitingRoom; Yellow = PlayerManager; Teal = MatchManager; Light yellow = program logic.

*Lines:* Purple = initiation; yellow = messages with ConfigServer; blue = listening loop for MS; red = message loop for MS; orange = match finding loop; green = player listening loop

**III. Coding Specifications**

**Coding Methodology:** The C# server will use the .NET Task-based Asynchronous Pattern (TAP) with the Async/Await keywords. This pattern uses Task objects, which are executed with the built-in .NET thread pool. Synchronization will thus be necessary for any shared data.

**Data Structures:** The following criteria is needed for storing waiting players:

-Add to end of the collection (new player comes in and waits at the end of the line)

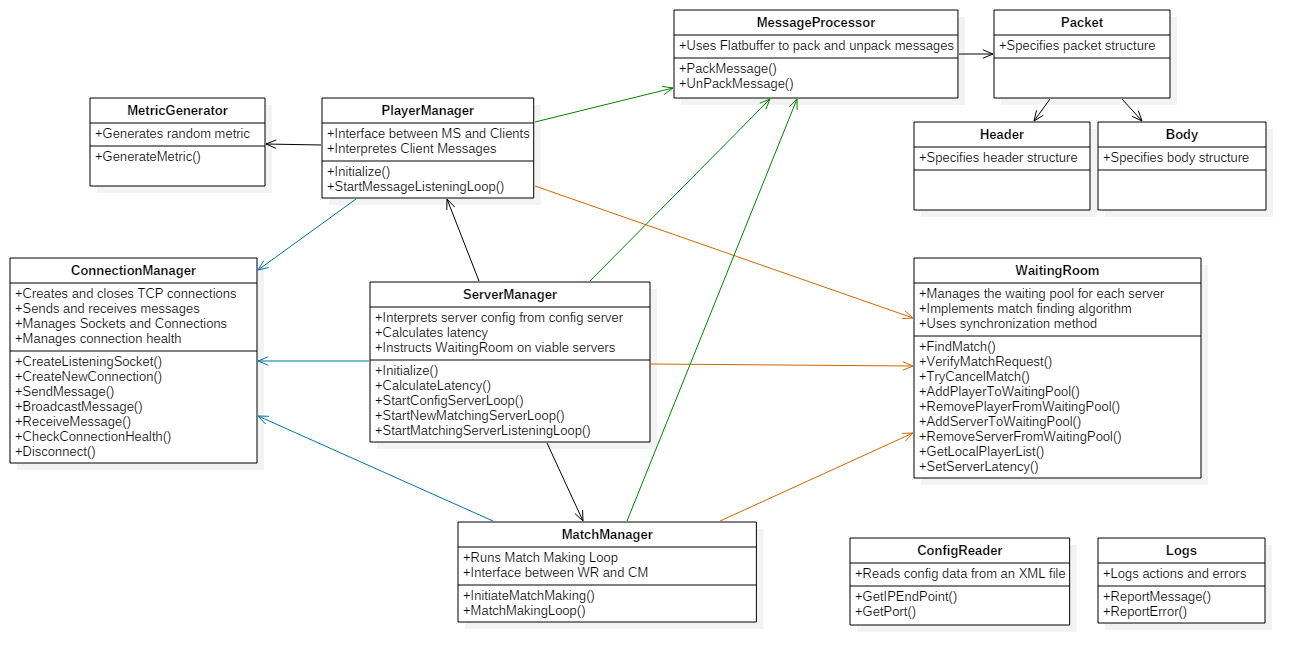
-Retainment of addition ordering (emulating wait time)

-Out of order deletion (remove matched players at front and any other position)

*Choice:* An OrderedDictionary is chosen as the container for waiting players as it facilitates rapid look-up and removal as well as time-relevant ordering, which is important for matching fairness in our algorithm.

*Synchronization:* The data structures will be managed in a ~~Singleton~~ class, WaitingPool, whose public methods will be synchronized by a lock. (NOTE: Using SynchronizationAttribute on the entire class was also considered, but unclear research suggests that this choice would have poorer performance).

**Proposed Class Diagram:**



**Note:** Line Color represents possession of an instance of a particular class (blue – ConnectionManager; green – MessageProcessory, orange – WaitingRoom).

**Classes:**

**ServerManager**

*Public Methods:*

**bool Initialize ()**

Function: Start the matching server program. Creates a connection with the ConfigServer and obtains the server’s ID code. It then starts four program loops: ConfigServerLoop, MatchingServerListenLoop, PlayerManagerLoop, and MatchingManagerLoop.

Return: This method returns true if the program was started without fail and false if a critical initialization component failed.

*Private Methods:*

**void StartConfigServerLoop ()**

Function: Loop asynchronous receive calls to the ConfigServer in order to receive messages from the ConfigServer.

**bool StartMatchingServerListeningLoop ()**

Function: Initiates listening for MatchingServers and then starts a loop with asynchronous accept calls to receive new connections from new MatchingServers.

Return: This function returns false if the listening socket creation failed, otherwise returns true;

**void StartNewMatchingServerLoop ()**

Function: Loop asynchronous receive calls on a new MatchingServer connection in order to receive messages from that MatchingServer.

**float CalculateLatency ()**

Function: Calculate a fake latency score for a connection with a particular server and call WaitingRoom.SetServerLatency() to set it in the WaitingRoom. This method is called occasionally in a MatchingServerLoop.

**ConnectionManager**

enum ConnectionType {

None = 0,

ConfigServer,

MatchingServer,

ConnectionServer,

Client

}

ConcurrentDictionary<string, Socket> socketList;

*Public Methods:*  
**bool CreateNewConnection (ConnectionType type, IPAddress address)**

Function: Creates a new connection of a specified type with a specified address.

Return: The method returns true on success and false on failure.

**bool CreateListeningSocket (ConnectionType type)**

Function: To create listening sockets for a specified connection type.

Return: The method returns true on success and false on failure.

**void AcceptNewConnection (ConnectionType type, out string connectionID)**

Function: Ask the connection server for an asynchronous accept call on particular listening socket of given type.

**bool SendMessage (****ConnectionType type, string sendToID, byte[] message)**

Function: Takes the sendToID, looks up the socket and sends the message to that socket.

Validation: The string should never be empty and the message should always be a minimum of the header size.

Return: The method returns true on success and false on failure.

**bool ReceiveMessage (ConnectionType type, string receiveFromID, out byte[] message)**

Function: Looks up the socket from the ID and awaits the receipt of a message on that socket. The message is passed as an out parameter. Failure informs the caller that the connection with the ID is no longer viable.

Validation: The string should never be empty.

Return: The method returns true on success and false on failure.

**bool BroadcastMessage (ConnectionType type, byte[] message)**

Function: To send a message to all MatchingServers

Return: The method returns true on success and false on failure.

**bool CheckConnectionHealth (ConnectionType type, string connectionID)**

Function: To check if a connection is still viable via a health check mechanism.

Return: The method returns true if the connection is healthy and false the connection is unresponsive.

**void Disconnect (ConnectionType type, string connectionID)**

Function: Tell the ConnectionManager to shut down a connection that isn’t needed anymore.

**PlayerManager**

*Public Methods:*

**bool Initialize ()**

Function: Public interface to start the player management loop.

Return: The method returns false if the client listening socket could not be made (OR) the connection with the ConnectionServer cannot be made. Otherwise, the function returns true.

*Private Methods:*

**void StartMessageListeningLoop ()**

Function: Initiates a loop asynchronously listening for messages from either the ConnectionServer or from clients (in the latter case, the loop waits for accept calls and upon accepting and creates new asynchronous loops for continual listening to each particular client.

**MetricGenerator**

*Public Methods:*

**float GenerateMetric ()**

Function: Creates a random metric score.

Return: A float value representing the metric score.

**MatchManager**

*Public Methods:*

**void InitializeMatchMaking ()**

Function: Public interface to start the matchmaking process.

*Private Methods:*

**void MatchMakingLoop ()**

Function: Asynchronous match making loop.

**WaitingRoom**

struct PlayerIdentification {

string matchingServerID;

string playerID;

}

enum MatchState

{

UNMATCHED = 1,

MATCHED\_BUT\_UNCONFIRMED,

CONFIRMED\_WITH\_SERVER

}

struct PlayerInfo {

PlayerIdentification identification;

MatchState state;

float metric;

}

struct Match {

PlayerIdentification playerOne;

PlayerIdentification playerTwo;

}

struct MatchingServer {

OrderedDictionary<string, PlayerInfo> playerQueue;

float latency;

}

Dictionary<string, MatchingServer> serverList;

object waitingRoomLock; // All public functions call the lock and all public functions DO NOT call any other public functions

*Public Methods:*

**bool FindMatch (out bool isLocalMatch, out** **Match match)**

Function: Runs the match finding algorithm.

Return: The method returns true on a successful match find and false on failure.

**bool TryCancelMatch (****Match match)**

Function: Attempts to cancel a match from a match cancel request or from a match request that was denied by another MatchingServer.

Return: The method returns true on successfully cancelling a match find and false on failure.

**bool VerifyMatchRequest (Match match)**

Function: Checks to see if another MatchingServer’s match request is viable and sets the states of the match accordingly.

Return: The method returns true on a successful verification and false if the match in question is not viable or the local server’s player has disconnected.

**bool AddPlayerToWaitingRoom (PlayerInformation player, float metric)**

Function: Adds a player to the waiting queue of a particular MatchingServer.

Return: The method returns true on a successful add and false if the addition could not happen (out of memory).

**void RemovePlayerFromWaitingRoom (PlayerInformation player)**

Function: Removes a player from the waiting queue of a particular MatchingServer if an entry exists.

**bool AddServerToWaitingRoom (string serverID)**

Function: Creates a new OrderedDictionary to hold the waiting queue of a new MatchingServer.

Return: The method returns true on a successful creation and false on failure.

**void RemoveServerFromWaitingRoom (string serverID)**

Function: Clears all data pertaining to a waiting queue of a particular MatchingServer, if it exists.

**bool GetLocalPlayerList (out PlayerInfo[])**

Function: Produces a list of the local MS’s players to send to a new MatchingServer, per request.

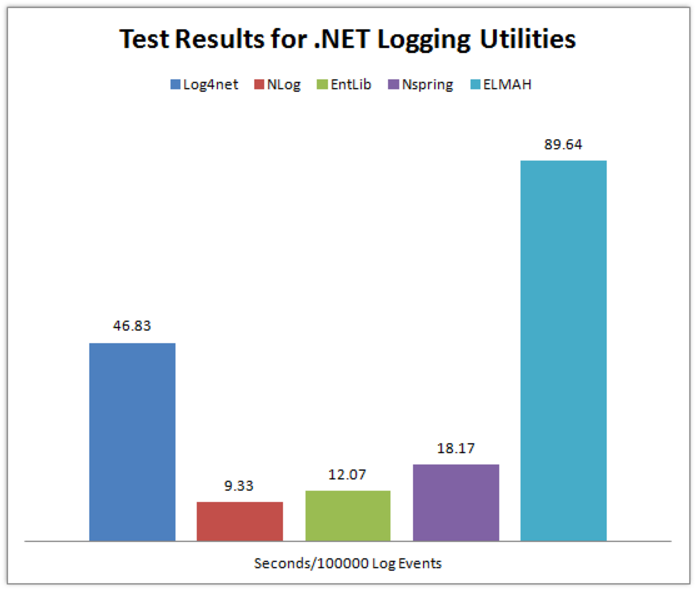
Return: The method returns true on a successful creation or false if the list is empty.

**void SetServerLatency (string serverID)**

Function: Sets the server latency value of a particular serverID.

**Logger**

*Commentary:* Creating a logging class to wrap our chosen logging library in case of need to change it later without changing any other code. Chose NLog over log4net due to presumed higher performance:



Ref: (<https://www.loggly.com/blog/benchmarking-5-popular-net-logging-libraries/>)

*Public Methods:*

**void ReportMessage (string message)**

Function: Sends a message to be logged by our logger.

**void ReportError (string errorMessage)**

Function: Sends an error to be logged by our logger.

**ConfigReader**

Network configuration file example:

<?xml version="1.0" encoding="utf-8"?>

<ConnectionInformation>

<ConfigServer>

<ip>10.100.10.6</ip>

<port>14040</port>

</ConfigServer>

<ConnectionServer>

<ip>10.100.10.6</ip>

<port>8433</port>

</ConnectionServer>

<MatchingServers>

<port>12000</port>

</MatchingServers>

<Clients>

<port>9992</port>

</Clients>

</ConnectionInformation>

*Public Methods:*

**static bool GetIPEndPoint (string xmlNodeName, out IPEndPoint endPoint)**

Function: This method retrieves the IPEndPoint of a connection entry in the network.xml configuration file.

Returns: Returns true upon a successful read and false if the data could not be obtained.

\*\*\*Special Comments\*\*\*

In the case of ConfigServer going down and going back up along with reconnection of a MS to the ConfigServer, there may be complications with the current assignment of MS ID made by the ConfigServer. For example, if the ConfigServer dies and restarts and the MS reconnects, that MS will get a different ID than previously acquired. The original ID is tied to all MS data structures and changing this value would be quite inconvenient. However, the current MS ID assignment system is just for simplicity. Later, the unique ID per MS would be predetermined and tied to verification protocol between the MS and the ConfigServer. The same MS connecting from the same IP and correct verification will be provided the same ID and no change of ID across all servers would be necessary.