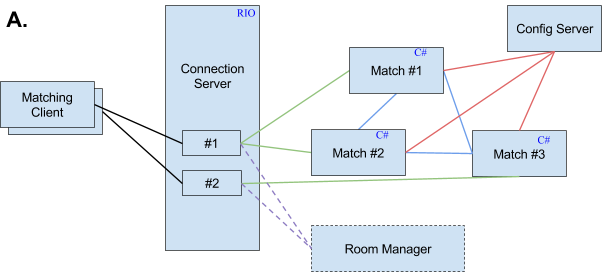
**Proposed Alternative Matching Server (C#) for 4:33 Network Engine Project**

**Mik Minier**

**Proposal Version 1.0.0.1**

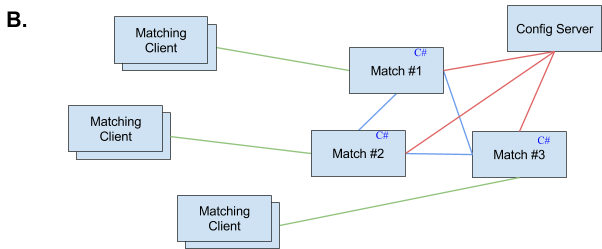
|  |  |  |  |
| --- | --- | --- | --- |
| Version | 날짜 | 변경 내역 | 이름 |
| 1.0.0.0 | 2016.12.22 | 초안 | 밐 |
| 1.0.0.1 | 2016.12.23 | +Data Structure/Synchronization Discussion | 밐 |

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



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.



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.

**Matching Server Function Specifications:**

**I. Index:**

**1. Server Connectivity**

-> Configuration Server

-> Other Matching Servers

-> Connection Server

**2. Player Connectivity**

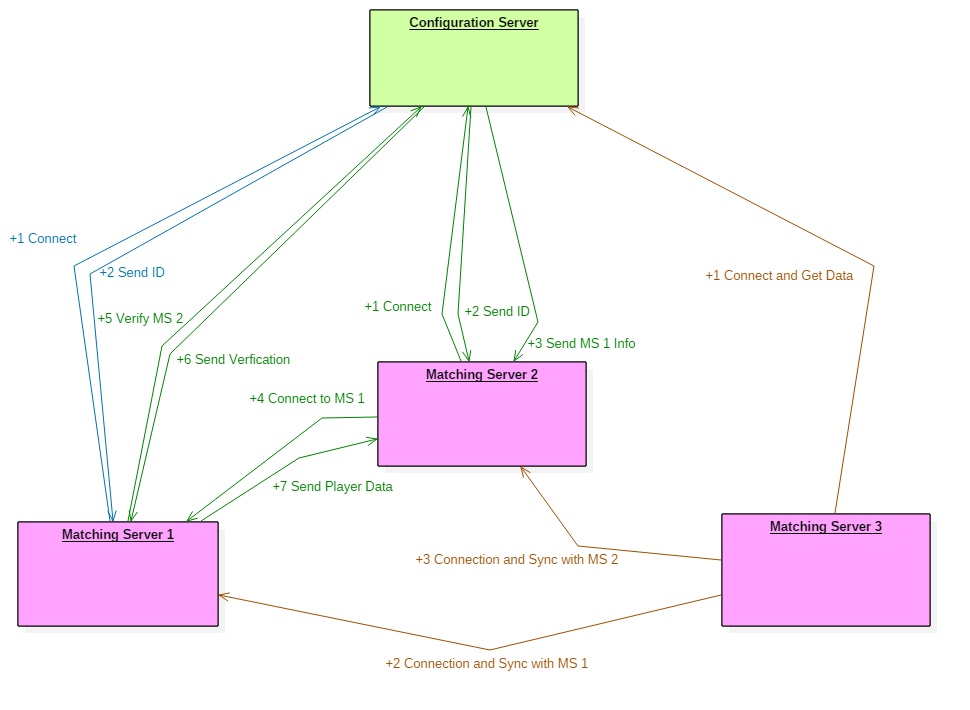
-> Through client directly

-> Through Matching Servers

-> Through Connection Server

**3. Match Making**

-> Rules and iteration methods



**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 connects to the ConfigServer and awaits to receive a message

[**FailCase**] The MS retries connection to ConfigServer

-2 The ConfigServer verifies/registers the MS and sends the MS its ID and the number of previously active MS

[**FailCase - disconnect**] The MS retries connection to ConfigServer

[**FailCase – socket error**]

The MS closes

-3 If the number of previously active MS is less than 1, the MS initializes its processes, otherwise it synchronizes

[**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

**II. Programming strategies and data structures:**

**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)

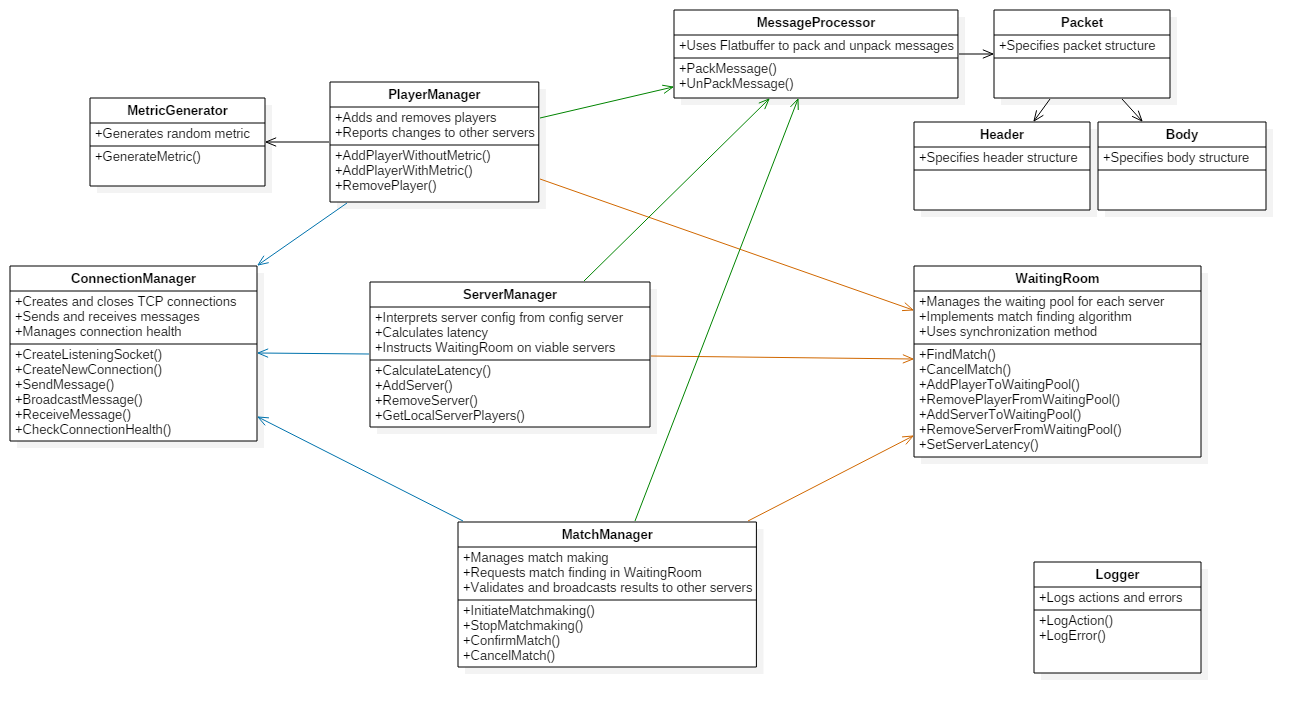
-Retainment of addition ordering (emulating wait time)

-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:**



enum MatchState

{

UNMATCHED = 1,

MATCHED\_BUT\_UNCONFIRMED,

CONFIRMED\_WITH\_SERVER

}

struct PlayerInfo

{

public float metric;

public int serverID;

public MatchState state;

public string matchedID; // ID of the player this particular Player is matched to

}

string uniquePlayerID = serverID.ToString() + “.” playerValue.ToString();

OrderedDictionary<string, PlayerInfo> serverWaitingPool;