Xbox One MatchSim

Summary

MatchSim is a server-based tool that allows game developers to see the effectiveness of their matchmaking hoppers and rules before releasing to real users. In the past, game developers could only perform tests of their matchmaking configuration through large scale betas with actual users. MatchSim allows developers to create large sets of match tickets and see which tickets are matched with each other. Developers can further iterate on this process by modifying their hoppers and rules, re-running the simulation, and seeing if the resulting matches are more preferable.

This document assumes background in the Xbox One Matchmaking system. For further details please see the XDK documentation.

Getting Started

MatchSim is a server-based tool. It accepts tickets through a HTTP request, and uses certificate based authentication to restrict access to only your title’s sandbox. As such, the first step for getting started is to obtain a Business Partner Cert, with the Matchmaking policy enabled. This is obtained through Web Service Management on XDP. See Appendix D for details on walking through XDP, assigning your web service permissions, and creating your .pfx and .cer file.

The MatchSim Client Tool

MatchSim has a REST endpoint, as listed below:

POST <https://smartmatchsim.xboxlive.com/matchsim/simulate>

It is possible to call this endpoint directly, using the source code in Appendix A. However, an associated command-line tool has been provided to make this easier. To run this tool, unzip it, place the .CER file in the same folder, and modify *MatchSimTool.exe.config* to use the .CER file you have created from the step above.

<appSettings>

<add key="MatchSimEndpoint" value="https://smartmatchsim.xboxlive.com/matchsim/simulate" />

<!-- The pfx for this cert must be installed for use. In addition this program must be run with administrator privileges. -->

<add key="CertificateFile" value="[Replace\_with\_your\_certificate\_file.cer]" />

</appSettings>

(Note that the tool must be run in an elevated command prompt to access the certificate.)

The MatchSim tool takes two arguments on its command-line. The first is a switch to determine whether it is generating a request, or sending a request. The second is an input file.

>MatchSimTool.exe generate sampleGenerator.xml

>MatchSimTool.exe send sampleRequest.txt

In general, you will want to create a small XML document explaining the kinds of tickets you want to create. Then you will call MatchSimTool with the **generate** command to create the JSON body, and save this to a file. You can inspect or modify these items, and when satisfied, you will call the MatchSimTool again, but with the **send** command, and with the JSON body as its argument. It will take care of calling the MatchSim service at the given endpoint, and report the match results.

Alternatively, you can craft the JSON request by hand—the **generate** command simply provides a shortcut for creating the JSON body.

Samples of both XML document and the JSON body are provided in Appendix B and C. Note that the **send** sample will not run directly, as your sandbox will have both a different certificate and a different hopper configuration. Further details of the request format is described below.

Input file for Generate command

The MatchSim tool takes a fairly simplified XML document when run in generate mode. There are three areas of interest:The title configuration section, the ticket configuration section, and the various attributes for each ticket that is created. Each of these will be described in order.

# Title Configuration

<TitleConfiguration>

<sandbox>SLSG.91</sandbox>

<scid>27220100-ff39-4d5c-8d29-e045123b0933</scid>

<hopper>MatchSim\_Sample\_Hopper</hopper>

<templateName>SampleTemplate</templateName>

</TitleConfiguration>

The title configuration section is a set of elements describing your title. These are your sandbox, scid, hopper, and session template name.

# Ticket Configuration

<giveUpDuration>25</giveUpDuration>

<numberOfTickets>2</numberOfTickets>

<playersPerTicket>1</playersPerTicket>

<timeBetweenTicketsInMs>200</timeBetweenTicketsInMs>

The ticket configuration section describes long your tickets wait before expiry(in seconds), how many tickets will be created, and how much time passes between each ticket’s arrival.

# Attribute Configuration

<playerAttributes>

<attribute type="integer" name="targetRank" min="0" max="50"/>

<attribute type="double" name="skill" min="0" max="5"/>

</playerAttributes>

<ticketAttributes>

<attribute type="string" name="desiredMap" options="HighRoad,LowRoad,MiddleRoad"/>

</ticketAttributes>

<QoSAttribute>true</QoSAttribute>

<ServerQoSAttribute distributionType="randomPingTime">

<server name="server1" minPing="10" maxPing="500" />

<server name="server2" minPing="10" maxPing="500" />

</ServerQoSAttribute>

Each ticket will have a set of attributes applied to it. These may be player attributes(one per player within the ticket) or ticket attributes(one applied to the whole ticket). The generator is given the bounds of the attribute, and will randomly pick a value within those bounds to apply to each ticket.

For the ServerQoSAttribute tag, note the distributionType may be of the following options:

* None—No server QoS attribute provided
* Fixed—All tickets receive the server measurements:server1:50, server2:150
* randomPingTime—All tickets receive random pings based on minimum and maximum values, as per the example above.

After execution, the resulting output will be a JSON body, described below, which can also be used in the send command.

Input for Send command

The MatchSim tool takes a text file of JSON when in send mode. This JSON body consists of configuration information, followed by a list of tickets. Below is a breakdown of the pieces of the request. See Appendix B for a complete JSON request.

The first few elements of the JSON input are a configuration section of three elements:

"SandboxID": "SLSG.91",

"ServiceConfig": "27220100-ff39-4d5c-8d29-e045123b0933",

"HopperName": "Sample\_Hopper",

Replace these with your sandbox, serviceConfig, and the hopper you wish to match in. This hopper is set up on XDP, and can be the actual hopper currently in use for matchmaking in RETAIL. It can also be the one in your development sandboxes that you have allocated for test purposes. Note that MatchSim cannot modify these rules—it merely uses ones that have been created separately.

The next element is a TicketCollection of TicketArrivals. A TicketArrival represents one CreateTicket Request that would ordinarily be made to the match service, along with its arrival time. A TicketCollection is a JSON array of these arrivals, as shown below.

"TicketCollection": {

"TicketArrivals": [{

"ArrivalOffset": 0,

"Xuid": 1000,

"Ticket": {

"scid": "27220100-ff39-4d5c-8d29-e045123b0933",

"hopperName": "Sample\_Hopper",

"giveUpDuration": 25,

"ticketSessionRef": {

"scid": "27220100-ff39-4d5c-8d29-e045123b0933",

"templateName": "SampleTemplate",

"sessionName": "1faebd76-b276-4a89-b24b-96760b0d976a"

},

"preserveSession": “Never”,

"players": [{

"xuid": "1000",

"playerAttributes": {

},

"playerTicketAttributes": {

}

}],

"ticketAttributes": {

},

"internalTicketAttributes": {

}

},

"ServerMeasurements": []

},

[… more ticket arrivals may follow]

Each of these elements is described below in order of their appearance:

|  |  |
| --- | --- |
| Attribute | Description |
| ArrivalOffset | The time from the start of the simulation, in milliseconds, that the ticket is expected to be sent to the match service. |
| Xuid | The xuid of the user submitting the ticket. |
| Ticket scid | This should match the scid in the configuration section. |
| HopperName | This should match the hopper in the configuration section. |
| giveUpDuration | The length of time, in seconds, that the ticket remains active before expiring. |
| ticketSessionRef:scid | This should match the scid in the configuration section |
| ticketSessionRef:templateName | The session template associated with the hopper |
| sessionName | This should be a unique guid. |
| preserveSession | Either “Always” or “Never”. “Always” indicates this ticket is for a game in progress, whose session will be reused in the match. “Never” indicates a different session will be returned. The only implications for matching are that two tickets with “preserveSession”:”Always” cannot be matched with each other. |
| Players | This structure contains information about each of the players inside the ticket. |
| Players:xuid | The xuid of the user. Generally these should be unique integers. Tickets cannot match other tickets with the same xuid(a user does not want to match with himself). |
| Players:playerAttributes | Information describing players, such as their skill or preference. |
| Players:PlayerTicketAttributes | This should be a unique guid. |
| TicketAttributes | Information given by the ticket creator that applies to the ticket as a whole, as opposed to a particular user. An example might be the map desired by the group. |
| InternalTicketAttributes | Attributes which are normally filled in by the server, but can be set for MatchSim. For instance, peer-to-peer QoS prediction currently uses internalTicketAttributes to generate longitude/latitude for each ticket. These values are instead passed through internalTicketAttributes. |
| ServerMeasurements | Information used for a hopper configured with the clientServer QoS rule. This contains the ping information to dedicated servers. The following example describes a ticket whose console had recorded a ping of 50ms to server1, and 150ms to server 2:  [ {"Key":"server1","Value":50,"Key":"server2","Value":150} ] |

Note that some of these fields are duplicated. In a future release this duplication may be removed—for now it remains because it is most applicable to one of MatchSim’s original goals, which is replay of actual ticket traffic.

Request Output

Once a request has been made, the server will perform matchmaking on these tickets, using the specified hopper’s configuration. It then responds with a list of all matched ticket groups, a list of all ticket responses, and a collection of statistics. Each of these three sections is detailed below.

# Group List

"GroupList": {

"MatchedGroups": [{

"MatchTimeOffsetInMs": 0,

"Players": [{

"xuid": "1000",

"playerAttributes": {

},

"playerTicketAttributes": {

"PlayerSkill": 20

}

},

{

"xuid": "1020",

"playerAttributes": {

},

"playerTicketAttributes": {

"PlayerSkill": 18

}

}],

"MatchDistance": 0.2,

"TicketIDs": [

"ffe0548b-8683-4740-9793-ef2e55e54650",

"54b1a4ce-a5c2-4b36-9e5c-d452dae55d01"

],

"ServerCandidates": [

{

"Key": "server1",

"Value": 120

},

{

"Key": "server2",

"Value": 251

}

},

[…more matched groups]

}

Here, the players who were matched together are listed by xuid, along with their attributes, match distance, ticketIDs, and server candidates. Distance is calculated as a sum of the differences between the players’ attributes, scaled by the maximum allowed distance in the max generation, according to the hopper’s rules. Larger distances means a poorer match, while a distance of zero indicates identical attributes for the items compared.

For groups with more than one user, the distances are the differences of all tickets compared to the centroid of this group—the seed that was used to pull neighboring tickets together.

# Ticket Response List

"TicketResponseList": {

"Tickets": [{

"ticketId": "8ed115b3-db75-48b6-aebd-a0aff0b66d79",

"waitTime": 0

},

{

"ticketId": "8c8b9ba6-6db7-4e1f-958e-7247e738764c",

"waitTime": 0

},

}

The Ticket Response List contains all created tickets, along with the estimated wait time returned on ticket creation. TicketIDs are generated as random guids, while waitTimes are in seconds. Note that this estimated wait time is only what is given on ticket creation, and is different from the actual wait time.

# Statistics

"Stats": {

"TicketsProcessed": 1000,

"MinMatchingTimeInSeconds": 0,

"MaxMatchingTimeInSeconds": 0,

"AvgMatchingTimeInSeconds": 0.0,

"AvgGroupSize": 2.0,

"AvgMatchDistance": 0.0

"PlayersWithNthChoiceServer": "392,59,29,16,0,0,0,0,0,0",

"PlayersWithServerNotChosen": 0

}

The final section is an aggregated set of statistics. These statistics give a general feel for how long the matches took, how large the groups were, and the quality of the matches, measured by the distance of the tickets. Note that matching time is aggregated per ticket(as opposed to per player). For configurations that use Client/Server QoS, the statistics also report how many players received their 1st place server, 2nd place server, and so on.

At a glance, these statistics tell how effective your hopper configuration is at matching the input set given.

# Suggested Use

Once you can generate requests, the next step is to create them with a purpose in mind. Below are three such suggested uses for calling MatchSim.

**Education**—While the XDK documentation gives a description of how hoppers and rules interact, the clearest way to understand the meanings of your rules is through experimentation. For instance, to understand the effect rules have on match quality(measured by distance), you may wish to create MatchSim requests of just the minimum number of people required to form a group. Then you can vary the attributes to see which combinations will be perceived as better.

**Iteration**—Once you understand the rules and how they work, you will want to create a representative sample of what you expect your population will be like. Then you can simulate how they will match, and iterate over your configuration until you have optimized for your desired balance of time and quality.

**Replay**—After your game has released, your population will naturally be different than predicted, or may perform in unexpected ways. By taking a sample of actual traffic and replaying it, you will be able to understand why your matches occurred, and modify your configuration to rectify any issues.

# Known Issues

This tool is still in development—as such, there are known issues that we’ll work be working on.

* Distance calculation is not accurate in max generation cases, for String and Collection Should rules. This does not affect who matches whom, provided they match before the max generation. The distance MatchSim reports back to you will be inaccurate, however.
* QoS Must rules require a nat type in the PeerToPeer and PeerToHost options—however, this cannot be specified at this time, so these QoS Must rules cannot currently be used with MatchSim. QoS Should rules are unaffected.

# Troubleshooting

**I receive a 401 in response, with no information**

This is most likely a certificate issue. Ensure:

1. Your Business Partner Certificate is installed.
2. The name of the certificate in the config file(MatchSimTool.exe.config) has your Business Partner Certificate’s name
3. MatchSim is run in an elevated command prompt—if it’s not, then it may not have access to the cert.

**I receive the following:”StatusCode:Forbidden Reason:Forbidden The request is attempting to access a scid/sandbox/hopper combination that the certificate does not have access to.”**

This is most likely an issue with your configuration. Ensure that your configuration has been published after creating the web service and certificate, and that the web service has been given appropriate permissions.

# Version History

v1.22

* Updated serverQoSAttribute to contain multiple possibilities. “fixed” and “none” replace the old values of true and false, while a new “randomPingTime” allows randomized server pings per ticket, based on specified minimum and maximums.
* Updated output to return ticketIDs in each matched group, as well as report better statistics about client/server QoS results. Stats now report how many players received their best choice, second best choice, and so on.

v1.21

* Updated to add an optional flag for the send command, “overrideHopper [hoppername]”. When set this replaces all hopper names with the given override, allowing a single ticket file to be easily sent to separate hoppers.
* In generate command, fixed a bug where scids were random.

v1.2—Updated to allow use of certificates generated per-sandbox.

v1.1—Add a <serverQoSAttribute> element to the generator xml. When set to true, creates a default, fixed server QoS measurement.

Appendix A—Sample C# code to send request with certificate

// Note:This must code must be run with access to the certificate store.

private static HttpWebRequest CreateWebRequest(string data)

{

byte[] requestBytes = Encoding.UTF8.GetBytes(data);

HttpWebRequest webRequest = (HttpWebRequest)WebRequest.Create(SimulateLoc);

if (ClientCertificate != null)

{

webRequest.ClientCertificates.Add(ClientCertificate);

}

webRequest.Method = "POST";

webRequest.ContentType = "text/html";

webRequest.ContentLength = requestBytes.Length;

using (Stream webpageStream = webRequest.GetRequestStream())

{

webpageStream.Write(requestBytes, 0, requestBytes.Length);

}

return webRequest;

}

Appendix B—Complete Sample of input in Generate mode

<Request>

<TitleConfiguration>

<sandbox>SLSG.91</sandbox>

<scid>27220100-ff39-4d5c-8d29-e045123b0933</scid>

<hopper>Sample\_Hopper</hopper>

<templateName>SampleTemplate</templateName>

</TitleConfiguration>

<giveUpDuration>25</giveUpDuration>

<numberOfTickets>2</numberOfTickets>

<playersPerTicket>1</playersPerTicket>

<timeBetweenTicketsInMs>200</timeBetweenTicketsInMs>

<playerAttributes>

<attribute type="integer" name="targetRank" min="0" max="50"/>

<attribute type="double" name="skill" min="0" max="5"/>

</playerAttributes>

<ticketAttributes>

<attribute type="string" name="desiredMap" options="HighRoad,LowRoad,MiddleRoad"/>

<attribute type="collection" name="dlcPacks" options="firstPack,secondPack,thirdPack,fourthPack"/>

</ticketAttributes>

<QoSAttribute>true</QoSAttribute>

<ServerQoSAttribute distributionType="none" />

</Request>

Appendix C—Complete JSON of input in Send mode

{

"SandboxID": "SLSG.91",

"ServiceConfig": "27220100-ff39-4d5c-8d29-e045123b0933",

"HopperName": "Sample\_Hopper",

"TicketCollection": {

"TicketArrivals": [{

"ArrivalOffset": 0,

"Xuid": 1000,

"Ticket": {

"scid": "27220100-ff39-4d5c-8d29-e045123b0933",

"hopperName": "Sample\_Hopper",

"giveUpDuration": 25,

"ticketSessionRef": {

"scid": "27220100-ff39-4d5c-8d29-e045123b0933",

"templateName": "SampleTemplate",

"name": "42d51a04-231a-4cee-be6c-6523007c47dd"

},

"preserveSession": "Never",

"players": [{

"xuid": "1000",

"playerAttributes": {

"PlayerSkill": 10

},

"playerTicketAttributes": {

}

}],

"ticketAttributes": {

},

"internalTicketAttributes": {

"locationData": "-44.8486876836273,-69.6603588804884"

}

},

"ServerMeasurements": []

},

{

"ArrivalOffset": 200,

"Xuid": 1001,

"Ticket": {

"scid": "27220100-ff39-4d5c-8d29-e045123b0933",

"hopperName": "Sample\_Hopper",

"giveUpDuration": 25,

"ticketSessionRef": {

"scid": "27220100-ff39-4d5c-8d29-e045123b0933",

"templateName": "SampleTemplate",

"name": "1faebd76-b276-4a89-b24b-96760b0d976a"

},

"preserveSession": "Never",

"players": [{

"xuid": "2000",

"playerAttributes": {

"PlayerSkill": 3

},

"playerTicketAttributes": {

}

}],

"ticketAttributes": {

},

"internalTicketAttributes": {

"locationData": "-5.44514586005599,-12.8522508371865"

}

},

"ServerMeasurements": []

}]

}

}

Appendix D—Detailed steps for creating Business Partner Certificate in XDP

To create your .pfx file:

1. Go to XDP (<https://xdp.xboxlive.com>) and access your Web Service Management page (Generally, only users who are in Administration Group for a company).
2. Use the following PowerShell script (or use your own methods) to generate a CSP blob.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

# Generate a properly sized key and make a certificate request

$certRequest = new-object -ComObject X509Enrollment.CX509CertificateRequestCertificate

$certRequest.Initialize(2) # Initialize in the machine context

$certRequest.PrivateKey.Length = 2048

$certRequest.PrivateKey.ProviderName = "Microsoft Enhanced RSA and AES Cryptographic Provider"

# Set XCN\_NCRYPT\_ALLOW\_EXPORT\_FLAG

$certRequest.PrivateKey.ExportPolicy = 2

# Subject is requred by the tool even though it is overwritten when we generate the certificate

$subject = new-object -ComObject X509Enrollment.CX500DistinguishedName

$subject.Encode("CN=NOT USED")

$certRequest.Subject = $subject

# Stores private key to allow auto binding. Import the .cer file you get back into the Local Machine/Personal store to bind to the private key

$enroll = new-object -ComObject X509Enrollment.CX509Enrollment

$enroll.InitializeFromRequest($certRequest)

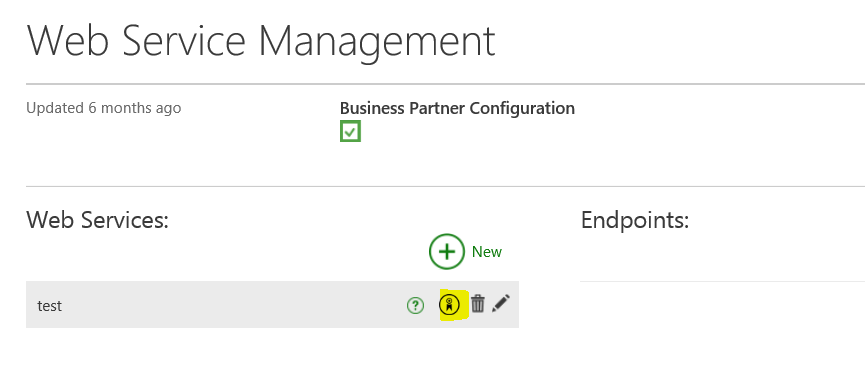
$strCert = $enroll.CreateRequest(0)

# Then export public key to be e-mailed

$certRequest.PrivateKey.Export("PUBLICBLOB", 0x40000001)

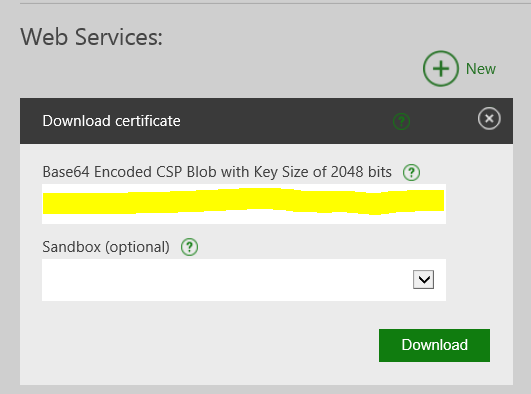
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

1. Click the following highlighted Icon and “Download certificate” is popped up.



**Note**—If you are creating this web service for this purpose, ensure it has been published before attempting to use the certificate.

1. Enter Step 2’s CSP blob in the following field and click Download and save .cer file on your PC.



1. Import .cer file at Step 4 through Microsoft Management Console.
   1. Open Microsoft Management Console (Win+R and enter “mmc.exe”).
   2. File > Add/Remove Snap-in… (Ctrl+M).
   3. Select “Certificates” and click Add in the middle of the window. On the wizard Select “Computer Account” and Click Next > Select “Local Computer” and Click Finish.
   4. Open Console Root > Certificates (Local Computer) > Personal > Certificates, and Action Menu > All Tasks > Import…
   5. On Certificate Import Wizard, click Next, click Brose to select the .cer downloaded at Step 4, click Next and click Finish.
2. Export the certificate imported at Step 5.
3. Continue to use Microsoft Management Console opened at Step 5.
4. Select the certificate installed at Step 5, and Action Menu > All Tasks > Export…
5. On Certificate Import Wizard, click Next and when “Export Private Key” page is displayed then select “Yes, export the private key”.
6. Proceed to select Next twice and when “Security” page is displayed then select “Password”. Enter Password and Confirm Password.
7. Proceed to select Next and on “File to Export” page, enter File Name select Next and Finish.

Once complete, you should install the cert’s .pfx file into your Personal store. (Note: the .pfx file can be installed to any PCs which you want to run MatchSim). Then, export this certificate from the certificate tool into a .CER file. You will need this .CER file to call MatchSim in the future.

Steps for importing the .pfx file:

1. Open Microsoft Management Console (Win+R and enter “mmc.exe”).
2. File > Add/Remove Snap-in… (Ctrl+M).
3. Select “Certificates” and click Add in the middle of the window. On the wizard Select “Computer Account” and Click Next > Select “Local Computer” and Click Finish.
4. Open Console Root > Certificates (Local Computer) > Personal > Certificates and Action > All Tasks > Import…
5. On Certificate Import Wizard, click Next, click Brose to select the .pfx downloaded the above step, click Next and click Finish.

Steps for exporting .CER file:

1. On Microsoft Management Console, select the certificate you imported above, Action > All Tasks > Export…
2. On Certificate Export Wizard, click Next, select Base-64 encoded X.509 (.CER)”

Finally, you’ll also need to ensure this certificate has access your title via the Matchmaking read policy. From XDP, go to Service Configuration->Access. From there you’ll see a list of services. If there is no policy present, add a Read-Only policy, designating the certificate you’ve created in the steps above.

