

©Copyright 2018 Luis Villegas. All rights reserved.

Matchmaking in Destiny

BY

Luis Villegas

(B.S. Computer Science, Florida International University, 2002)

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science
in the graduate studies program
of DigiPen Institute of Technology
Redmond, Washington
United States of America

Spring
2018

Thesis Advisor: Dr. Dmitri Volper
DIGIPEN INSTITUTE OF TECHNOLOGY

GRADUATE STUDY PROGRAM
DEFENSE OF THESIS

THE UNDERSIGNED VERIFY THAT THE FINAL ORAL DEFENSE OF THE
MASTER OF SCIENCE THESIS OF _____ LUIS VILLEGAS _____

HAS BEEN SUCCESSFULLY COMPLETED ON _____ [date] _____

TITLE OF THESIS: _____ MATCHMAKING IN DESTINY _____

MAJOR FIELD OF STUDY: COMPUTER SCIENCE.

COMMITTEE:

_____ [signature] _____
[name], Chair

_____ [signature] _____
[name]

_____ [signature] _____
[name]

_____ [signature] _____
[name]

APPROVED :

_____ [signature] _____ [date] _____
[name] date
Graduate Program Director

_____ [signature] _____ [date] _____
[name] date
Associate Dean

_____ [signature] _____ [date] _____
[name] date
Department of Computer Science

_____ [signature] _____ [date] _____
[name] date
Dean

The material presented within this document does not necessarily reflect the opinion of the Committee, the Graduate Study Program, or DigiPen Institute of Technology.

INSTITUTE OF DIGIPEN INSTITUTE OF TECHNOLOGY
PROGRAM OF MASTER'S DEGREE
THESIS APPROVAL

DATE: _____ [date] _____

BASED ON THE CANDIDATE'S SUCCESSFUL ORAL DEFENSE, IT IS
RECOMMENDED THAT THE THESIS PREPARED BY

LUIS VILLEGAS

ENTITLED

MATCHMAKING IN DESTINY

BE ACCEPTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE
DEGREE OF MASTER OF COMPUTER SCIENCE FROM THE PROGRAM OF
MASTER'S DEGREE AT DIGIPEN INSTITUTE OF TECHNOLOGY.

[Signature]

[Name]

Thesis Advisory Committee Chair

[Signature]

[Name]

Director of Graduate Study Program

[Signature]

[Name], Associate Dean

[Signature]

[Name], Dean of Faculty

The material presented within this document does not necessarily reflect the opinion of the Committee, the Graduate Study Program, or DigiPen Institute of Technology.

TABLE OF CONTENTS

Contents

ACKNOWLEDGEMENTS	6
ABSTRACT	7
CHAPTER 1: NTRODUCTION	8
CHAPTER 2: MATCHMAKING ARCHITECTURE	10
CHAPTER 3: MATCHMAKING CLIENT ROLE	12
CHAPTER 4: MATCHMAKING SERVER ROLE	23
CHAPTER 5: MATCHMAKING SKILL	28
CHAPTER 6: FUTURE WORK	49
BIBLIOGRAPHY	51

ACKNOWLEDGEMENTS

I wanted to thank Dr. Edward Kaiser for his patience being my advisor over the years and pushing me to make this thesis real. To Bungie I owe infinite gratitude for giving me the opportunity to lead the development of Destiny's Matchmaking system. The success of the system is owed to all the talented folks in the development and test teams at Bungie, and I thank them for being the best team I ever worked with. They are the heroes that kept pushing the envelope every day.

I also wanted to thank Digipen for the opportunity to finish my thesis over time while working full time.

Last but not least I wanted to thank my parents, my wife, and my kids for the words of encouragement and the never ending support.

ABSTRACT

Online multiplayer games have the need to group players together to facilitate competitive play and/or to minimize latency to improve the network experience. Grouping players automatically is referred to as “Matchmaking”, and this thesis outlines how such a system can be developed for an online multiplayer game. A history of matchmaking, high level architecture, and detailed explanations for how all the components fit together are presented. The thesis closes with an analysis of the outcomes of different matchmaking skill algorithms, and outlines different avenues for future work for advancing the matchmaking space.

CHAPTER 1: INTRODUCTION

The 1990's will forever be remembered as the decade when the world started connecting to the internet. In the early years of the decade the internet was still a hot commodity, with a penetration of 0.3% of the population (ITU, 2016). Even though internet access was rare, the games industry saw the incredible opportunity to connect players in different physical locations to share an entertainment experience. Doom (id Software, 1993), a popular first-person shooter released in the early part of the decade, enabled network gameplay by allowing different players to connect via an intranet. The process was tedious, the player needed to have a good understanding of their intranet setup, capabilities of the network, and port usage (id Software, 1995). Doom players were eager to grow the multiplayer experience from the intranet to the internet, and developed programs to send game data across the internet, enabling effective internet play (Coleman, 1995)

It was clear that sharing gaming experiences with other players through the internet was becoming a critical component to the future of gaming. id Software released another first person shooter in 1996 called Quake (id Software, 1996), which simplified internet online gameplay by using the concept of Server Browsers. The idea behind Server Browsers was to allow players to find game sessions to join via a global directory of available games. Players could register their session to a Server Browser to allow other players to find them. This paradigm became the de facto standard for the 90s, and several online games offered online gameplay via Server Browsers.

In early 2004, Bungie studios released a first-person shooter called Halo 2 (Bungie, 2004). Halo 2 evolved online gaming by introducing the concepts of

Matchmaking and skill-ranking to this genre. The idea behind Matchmaking was to automate the process of finding sessions to play with, providing a steady flow of games with different opponents with minimal manual intervention. Bungie worked closely with Microsoft defining the feature set needed for Matchmaking, and the final system was exposed by Xbox Live for other titles to use. Matchmaking then became the new standard for online gaming for player vs player matches.

Bungie released other titles in the Halo franchise using Xbox Live's Matchmaking services. In 2014 Bungie released *Destiny* (Bungie, 2014), a multi-platform first person shooter which expanded Matchmaking from player vs player matches, to include Matchmaking in different areas for player vs enemy activities. *Destiny* was released using Dementware's Matchmaking middleware services, a cross platform suite of services used for many Activision titles (Dementware, 2016).

In 2015, after the release of the major expansion to *Destiny* called *Destiny: The Taken King* (Bungie, 2015), Bungie started pursuing building its own cross-platform Matchmaking services. This project thesis is the documentation of the design and implementation of *Destiny*'s Matchmaking system, with emphasis on explaining the different challenges of such undertaking.

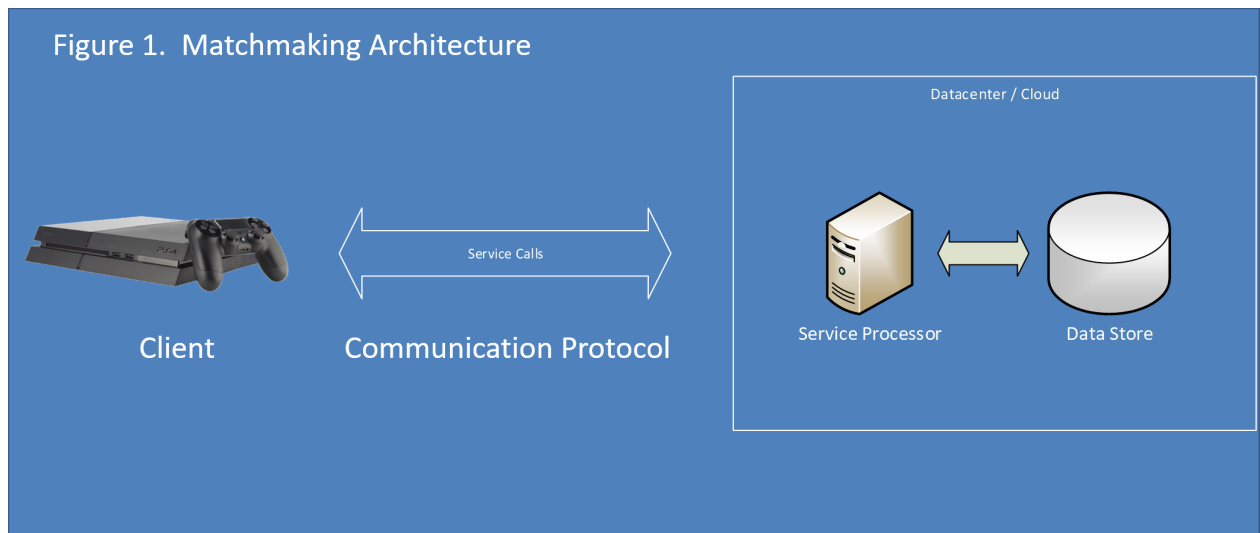
CHAPTER 2: MATCHMAKING ARCHITECTURE

Developing a Matchmaking system requires a top to bottom analysis to fully understand the responsibilities of each individual component. We need to establish our goals for the system first, and then decompose the system into different components that give us the flexibility to achieve those goals.

A Matchmaking system at the high level has the following goals:

1. Create automatic groupings of players for match/game participation.
2. Allow clients to share their properties and desires so optimal groupings can be formed.
3. Maintain enough global data about matches being formed and client properties to facilitate automatic group formation.

To achieve our goals, we are going to define different components and assign responsibilities to them (See Figure 1):



Client: This is the application or game that will participate in the Matchmaking process. The client is responsible for capturing the relevant properties of the match and the player and communicating those to a global system. The client is also

responsible for maintaining a local state machine to understand its progress in the Matchmaking process.

Service Processor: This is the component responsible for processing requests from the client. The requests include storing properties for all the clients participating in Matchmaking, evaluating matches, and returning pools of candidates for clients to consider joining.

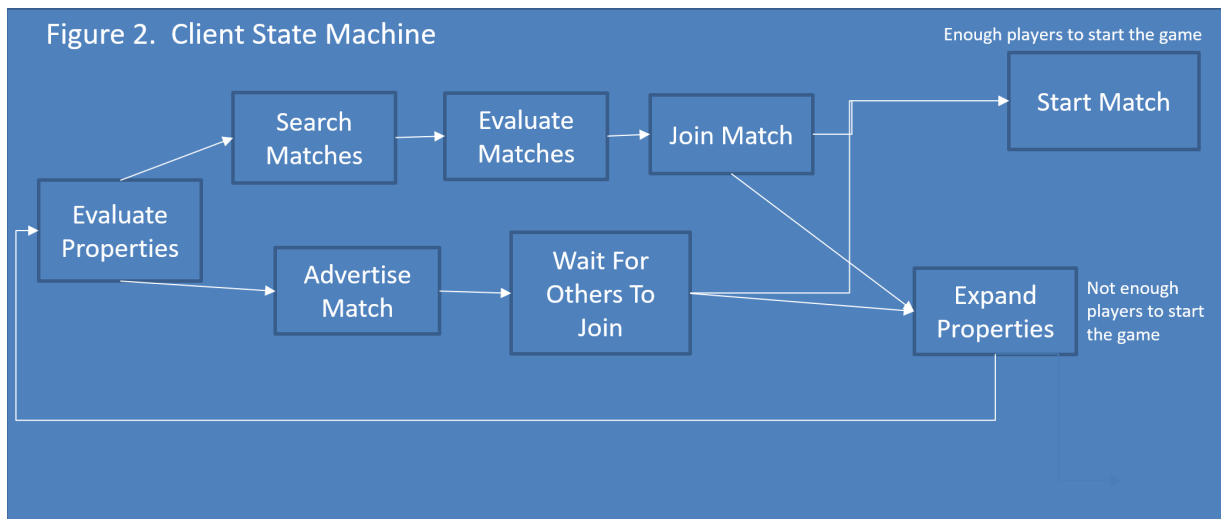
Data Store: This is a component used by the server to store the state of the Matchmaking system. This ensures flexibility for using diverse types of storage technologies when applicable. Separating this component also has an impact on scalability and fault tolerance, allowing the Data Store to live on a different Server, or on the Cloud, or in local storage if necessary.

Communication Protocol: This is a shared component between the client and the server, and it represents the contract for communication of properties and state changes of the Matchmaking system.

Note that most of the Matchmaking logic and the relevant state is performed by the client in this architecture.

CHAPTER 3: MATCHMAKING CLIENT ROLE

Now that we understand the Matchmaking system at the high level we will examine the different components and their behavior. We will start with the client and its relationship to the Service Processor. The Service Processor doesn't run a lot of logic, and the main responsibility it has is to evaluate matches on the clients' behalf and to keep a view of all the clients in the ecosystem. The client has two roles: searching and advertising. Searching refers to the idea of looking for matches that meet the client's criteria. Advertising refers to the idea of letting the system know that a match is available and that the client is waiting for other clients to join their match. The client has an internal state machine that dictates when it is time to search for matches, when to advertise the existing match, when to evaluate matches, and when to join another client. The following diagrams portray the logic that runs on the client in more detail:



The client's state machine goes through the following states:

Evaluate Properties: In this state the client decides to search for matches or advertise its current match. This state is subtle, but it is critical for the Matchmaking ecosystem. If all the clients were to choose to search at the same time nobody would be able to find

matches. If all the clients were to choose to advertise then there would be no clients searching for matches and no groupings would be formed. To tackle this challenge the system uses a probabilistic table to decide if the client needs to search or advertise. As clients go through the Matchmaking system the matches grow in size and they use a probability on a lookup table to determine which branch of the Matchmaking system they need to take. Here is one of the initial tables used in Destiny:

Table 1. Advertisement Chance

PARTY SIZE	ADVERTISEMENT CHANCE
1	16%
2	33%
3	50%
4	66%
5	83%
6	99%

Based on this table the system biases towards advertisement as matches become larger in size. This table assumes 6 vs 6 games, and it was hand tuned for the game in question, but a good rule of thumb for the advertisement probability of symmetric games with less than a couple of dozen players is to use $(\text{party size}) / (\text{max game size} / 2)$

Search Matches: In this state the client looks at the properties for all the players in the session and asks the Service Processor for matches that meet specified criteria. The properties in question are game specific, and can contain data such as party size, game type desired, skill for each player, average skill for the party, latency desired, etc. The Service Processor takes criteria and looks for matches in the Data Store that would be a

good fit for the client. A sample API level description of a request the client sends to the Server Processor for searching:

```
typedef struct _MatchmakingSearchRequest {
    bool has_SessionComposition;
    SessionCompositionData SessionComposition;
    bool has_ServiceConfigurationHash;
    uint32 ServiceConfigurationHash;
    bool has_SearchConfigOverrides;
    MatchmakingSearchConfigOverrides SearchConfigOverrides;
} MatchmakingSearchRequest;
```

Note that all the information for the session lives in the SessionCompositionData structure. The ServiceConfigurationHash is used to define which rule set to use and will be explained further in the Service Processor section, and the last field called “MatchmakingSearchConfigOverrides” is used for debugging and it allows the developers to change the behavior of the Service Processor for rapid iteration.

The SessionCompositionData structure is the one that expresses the properties of the client. In more detail:

```
typedef struct _SessionCompositionData {
    bool has_SiloId;
    uint64 SiloId;
    bool has_NatType;
    int32 NatType;
    size_t PartySizeComposition_count;
    int32 PartySizeComposition[32];
    bool has_GameInProgress;
    bool GameInProgress;
    bool has_BigPartyCount;
    int32 BigPartyCount;
    bool has_MaxAllowedJoinReservationCount;
    int32 MaxAllowedJoinReservationCount;
    bool has_LocalityInfo;
    LocalityData LocalityInfo;
    ...
}
```

Note that the data on this structure is game specific. On this example there is connectivity information (NAT type), game size information (PartySize, BigPartyCount,

LocalityInfo), and other miscellaneous fields. It is shown here for illustrative purposes as the contents of the structure will be different for different type of games.

Evaluate Matches: This is the most complex state of the state machine and it is responsible for getting a list of matches from the Service Processor and determining which are the best candidates to join. This state is split into subparts:

1. Sort list of results received from the Service Processor
2. Perform a quick connectivity test against each candidate in order
3. Sort the list again based on quick connectivity results
4. Perform a more involved connectivity test
5. Sort list again based on involved connectivity test results

To better explain the sorting process, it is important to understand the way matching criteria are expressed. For the Matchmaking system explained here there are two types of criteria: hard filter, and soft filter. Hard filters refer to properties that need to be exact during the matching process. Hard filters can be expressed as a binary expression or a range. Good examples here are game types (i.e. GameMode = “2vs2”) or exact ranges (i.e. Skill between “100 and 300”)

Soft filters refer to criteria that can have a weight applied to it. There are many criteria that could be evaluated via soft filters, and a good example is the skill rating. Via hard filters we can express a required range, and via soft filters we can give matches with skill close to ours a higher score.

When dealing with multiple soft filters we simply add the weights:

$$\sum_{j=1}^n w_j a_{ij}, \text{ for } i = 1, 2, 3, \dots, m.$$

Where a is the score for the property and w is the weight for the individual property. It is possible to have multiple soft filters, however adding more increases the complexity of the evaluation and it makes it difficult to understand the impact at the intuitive level.

Below is a detailed example for the evaluation of two soft filters. The first is skill and the second is latency with weights 40% and 60%, respectively. The soft filter is evaluated against the client properties. The evaluation process computes the soft filter score for each candidate, and this is what dominates the sorting process.

Table 2, 3, 4. Soft Filter example

Client Properties	Value
Skill	0.6 (-1000, 1000)
Latency	0.5 (0, 300)

Property	Weight
Skill	40%
Latency	60%

Candidate Properties	Value
Skill	0.7 (-1000, 1000)
Latency	0.75 (0, 300)

- $0.4 * |0.7 - 0.6| + 0.6 * |0.75 - 0.5|$
- $0.4 * |0.1| + 0.6 * |0.25| = 0.04 + 0.15$
- 0.19 -> Lower is better

After computing a score for each entry, it is possible to sort candidates in terms of desirability. Before moving on to the next stage of the evaluation process it is worth pointing that an important challenge with soft filters: balance. When using soft filters in production it is critical for engineering and design to be aligned on what the individual weights mean and the tradeoffs that are implied by changing the weights. We built a spreadsheet to help visualize the impact of the changes. The following tables show a general ideal of how to portray the tradeoffs made during soft filter balancing:

Table 5, 6. Soft Filter tradeoffs

Property	Weight
Skill	40%
Latency	60%

VS

Property	Weight
Skill	70%
Latency	30%

- Needs to be able to express tradeoffs in a meaningful way
- 100 skill points (5%) = 0.02 contribution
- 10ms = 0.02 contribution
- *A candidate with 80ms and 0 skill points delta gets the same score as a candidate with 40ms and 400 skill delta*

- 100 skill points (5%) = 0.035 contribution
- 35ms = 0.035 contribution
- 70ms change is worth a 200 (10%) skill expansion
- *A candidate with 175ms and 0 skill points delta gets the same score as a candidate with 35ms and 400 skill delta*

Latency Evaluation

Now that we understand the sorting process and the challenges of soft filters we will focus on latency evaluation. Skill algorithms carry a lot of complexity and are addressed in Chapter 5.

When evaluating latency, the algorithm presented uses a 3-phase process:


- **Phase 1:** Use the client’s IP address and map it to a physical location via Cartesian coordinates; this is referred to as “Locality” since it tries to capture the geographical location of the candidate, leveraging the fact that geographical closeness between two IP addresses tend to have a high correlation with lower latency. When the list of candidates is initially received there is no latency information and “Locality” is used for the initial sort.

To find the geographical location for a client we use a commercial database called “ip2location” that provides longitude/latitude information for millions

of IP addresses. Below is an example of the type of results provided by the conversion:

Table 7. Locality – ip2location

Player	Latitude	Longitude
BobSeattle	47.60621	-122.33207
MikeLA	34.05223	-118.24368

A map of the United States with a red line connecting Seattle, WA in the northwest to Los Angeles, CA in the southwest. The map shows state boundaries and labels for the two cities.

To simplify the evaluation of geographic distance we convert latitude/longitude into Cartesian coordinates using a base of 4,000 which is the radius of Earth in miles. The code to do so can be found below:

```
public static CartesianCoordinate AsCartesian(this GeoPoint geoPoint, float scale)
{
    double latitudeRadians = geoPoint.Latitude * Math.PI / 180;
    double longitudeRadians = geoPoint.Longitude * Math.PI / 180;

    var x = (float)(scale * Math.Cos(latitudeRadians) * Math.Cos(longitudeRadians));
    var y = (float)(scale * Math.Cos(latitudeRadians) * Math.Sin(longitudeRadians));
    var z = (float)(scale * Math.Sin(latitudeRadians));

    return new CartesianCoordinate(x, y, z);
}
```

The table below shows the conversion and accuracy for two geographical points derived from the IP address. One is in Seattle and the other one in Los Angeles. Once the latitude/longitude is converted to Cartesian coordinates the distance is computed which results in a close approximation to the actual geographical distance:

Table 8. Locality – Cartesian coordinates

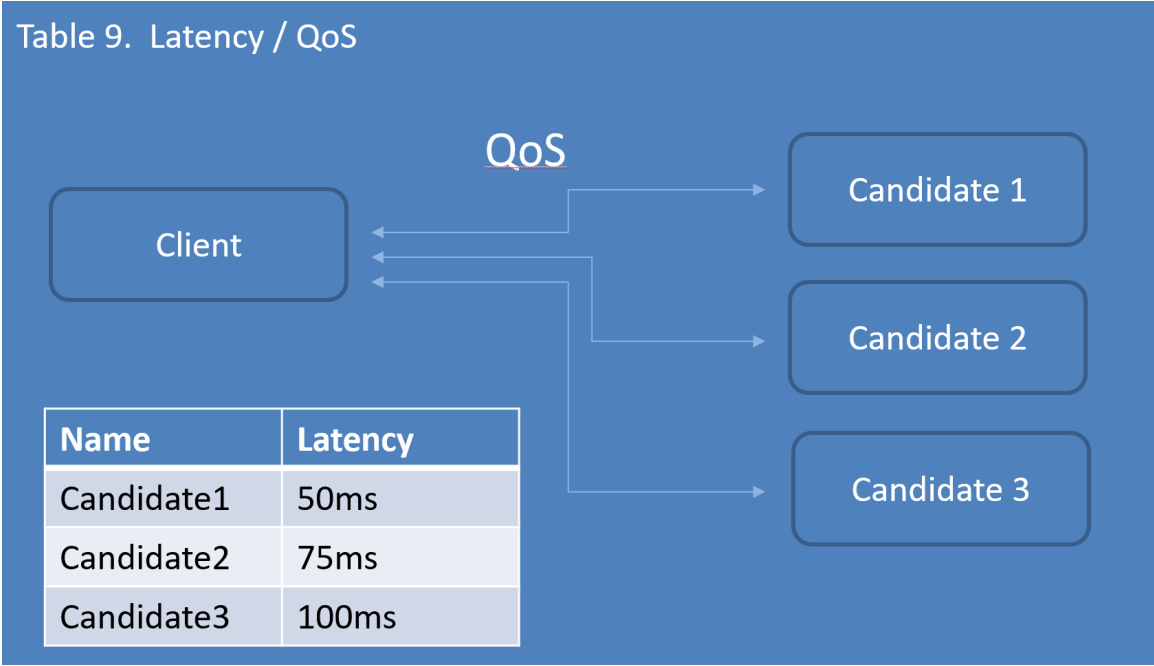
Player	Latitude	Longitude	LocalityX	LocalityY	LocalityZ
BobSeattle	47.60621	-122.33207	-1442.36487	-2278.502	2954.371
MikeLA	34.05223	-118.24368	-1568.311	-2919.542	2239.794

- $\sqrt{(x2 - x1)^2 + (y2 - y1)^2 + (z2 - z1)^2}$
- 970.71 → Close to the physical distance which is ~1000 miles

Locality evaluation of candidates via Cartesian coordinates is not perfect but it is a good starting point for the latency evaluation process. Note that there are some cases in which lower geographical distance is not correlated to lower latency, some examples are islands and locations separated by difficult geographical features that have made it difficult to build reliable internet infrastructure. In practice the later phases of the latency evaluation cull those candidates by measuring on the wire latency.

- **Phase 2:** Send a few packets with dummy payloads to the candidate list in sorted order to do a quick check on latency results, expecting candidates to reflect those packets back to use in a timely manner. Re-sort the candidates using the new latency information. From this point on Cartesian coordinates are ignored. If packets are not returned, then remove the candidate from the list. Note that clients that have connectivity issues due to cross-provider infrastructure challenges, router configuration problems, or network incompatibilities (i.e. moderate or restricted NATs) end up being removed

from the candidate list at this stage. The table below depicts how the searching client performs the latency checks and sorts the candidates.



- Phase 3:** After a configurable percentage of candidates have returned valid latency results from phase 2 we go back through the sorted list and do a more involved latency test. This new test sends packets to the candidates with specific game information to allow early rejection if the candidate and the game are no longer compatible. The results are re-sorted based on the new latency test results or removed from the candidate list if the candidate and searcher fail the compatibility test.

Once the Searcher has gone through all the 3 phases described above it goes through the sorted candidate lists and tries to join them in order. Once a successful join occurs the Searcher exits the evaluation process. In the event that joins are not successful or that there are no extra clients to evaluate the client expands the search parameters and starts the searching process all over again.

Expand Properties: This stage is responsible for expanding the search parameters for the client looking for candidates. The actual expansion is configurable, and it allows game developers to express how the matchmaking process should consider more candidates in the case that previous searches returned few/no acceptable results. This stage can be configured to ensure that eventually the matchmaking system converges by being more permissive over time. Expressiveness is important for this stage, and the following expansion illustrates this point:

- For the first 10 seconds find sessions with <100ms latency and within 100 points of my current skill
- After that for the next 30 seconds find sessions with <200ms latency and within 300 points of my current skill
- Afterwards find sessions with <500ms latency, and don't worry about skill delta

The goal is to allow different game modes to have different expansion rules. To achieve this goal, we use the idea of a ruleset. A ruleset is a set of expansion rules expressed through an xml file that lives on the server and is downloaded by the client during specific times depending on the game design (i.e. between matches), below is an example xml file for skill and latency expansions:

```
<ExpansionConfigs>
  <ExpansionConfig Name='skill_pvp_exponential' Type='skill'>
    <Expansion ExpansionDurationMs='10000' Value='150' />
    <Expansion ExpansionDurationMs='20000' Value='300' />
    <Expansion ExpansionDurationMs='20000' Value='600' />
    <Expansion ExpansionDurationMs='-1' Value='2000' />
  </ExpansionConfig>

  <ExpansionConfig Name='latency_fast_expansion' Type='latency'>
    <Expansion ExpansionDurationMs='5000' Value='100' />
    <Expansion ExpansionDurationMs='5000' Value='200' />
    <Expansion ExpansionDurationMs='5000' Value='300' />
    <Expansion ExpansionDurationMs='-1' Value='400' />
  </ExpansionConfig>
</ExpansionConfigs>
```

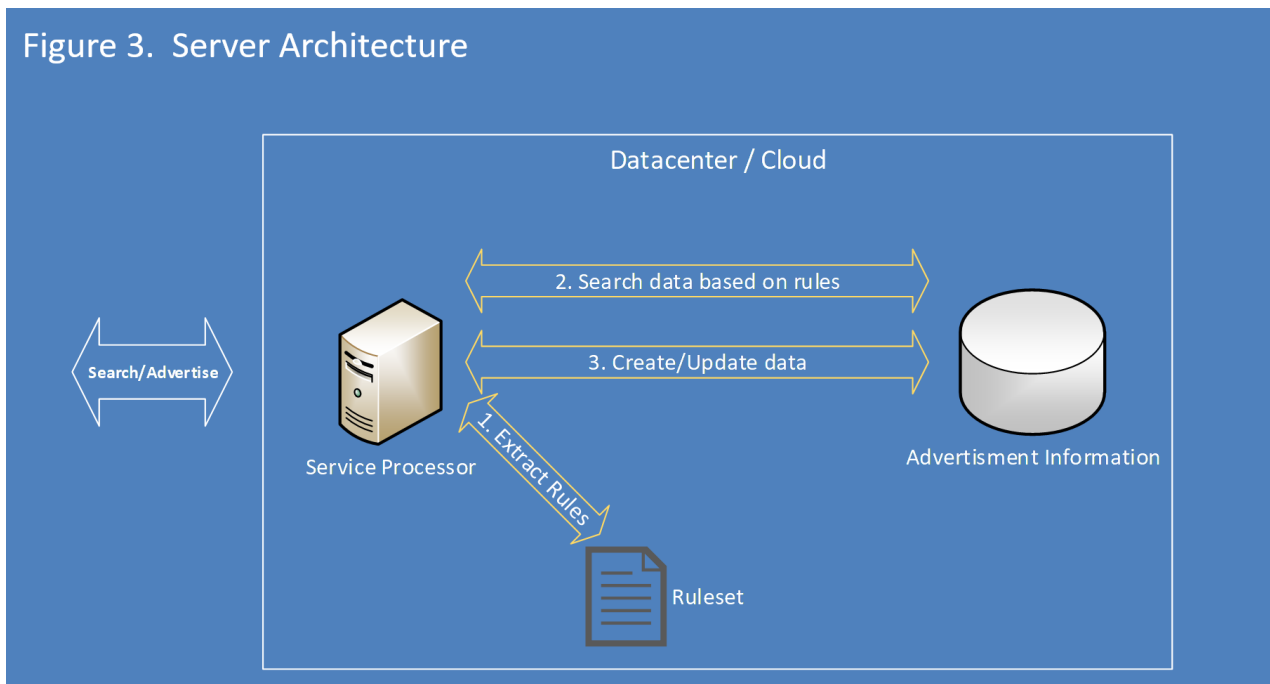
Having the ruleset files live on the server side allows game developers to change rules and deal with production changes without re-deploying the game or needing to create an updated version of the game to distribute to players.

Advertise Match: The client state machine has another branch for clients that want to advertise the game and wait for other clients to join them. This stage is simple, and at the core all it is doing is updating the parameters advertised and waiting for clients to join the match. Once the match has all the necessary players the state machine moves forward, and matchmaking no longer occurs. Advertisement publishes ranges for allowed criteria, and over time that criteria can be relaxed via the same mechanisms explained in the “Expand Properties” stage. Note that for a match a single client advertises on behalf of the group.

This section has gone over the logic that runs on the client and the various stages executed by the client’s state machine. The next section will focus on the logic that runs on the server side.

CHAPTER 4: MATCHMAKING SERVER ROLE

The responsibilities of the matchmaking server are very different than the responsibilities of the client. In the previous section we presented all the logic the client executes for the matchmaking process to be successful. The clients are stateful and move the state machine forward for matchmaking, keeping most of the complexity on the client. The server maintains less state about the process and instead is responsible for providing a central repository for all clients to interface with and to query and update data. Figure 3 shows the server architecture and the corresponding responsibilities:



For the server to be able to perform its responsibilities it provides a set of APIs for clients to be able to communicate with it. There are several technologies available to do this, ranging from custom protocols to more standardized solutions such as REST APIs and websockets. For the data being transferred via the APIs there are also several options such as JSON, XML, and other more compact solutions such as Google's protocol buffers. Given that web technologies keep evolving over time this paper will not focus

on any specific technology, and instead will present the choices used for implementation and the rationale behind the choices.

For Destiny the client needed to have a persistent TCP connection to the backend for progression and account calls. Matchmaking leverages the existing connection to implement the API calls made to the server. This reduced the need to create new connections to the server and unified the communication protocol for several systems. For the data protocol Destiny uses Google's protocol buffer (protobuf) because it provides a compact payload and has libraries for different languages. In Destiny's case the client is written in C++ and the server in C#, and the availability of protobuf libraries for both languages made it very convenient to adopt Google's data technology.

The following code snippets provide an idea of a structure defined in C# that would be transferred between client and server via protobuf (The C++ representation is similar):

```
typedef struct _MatchmakingCoreRequest {
    bool has_PlatformType;
    DeviceType PlatformType;
    bool has_RequestType;
    MatchmakingCoreRequestType RequestType;
    bool has_RequestNonce;
    uint64 RequestNonce;
    bool has_MatchmakingType;
    MatchmakingType MatchmakingType;
    bool has_MatchmakingSetId;
    uint32 MatchmakingSetId;
    bool has_ManagedSessionIndex;
    int32 ManagedSessionIndex;
    bool has_SearchRequest;
    MatchmakingSearchRequest SearchRequest;
    bool has_UpdateAdvertisementRequest;
    MatchmakingUpdateAdvertisementRequest UpdateAdvertisementRequest;
    bool has_DeleteAdvertisementRequest;
    MatchmakingDeleteAdvertisementRequest DeleteAdvertisementRequest;
    bool has_GetConfigurationRequest;
    MatchmakingGetConfigurationRequest GetConfigurationRequest;
} MatchmakingCoreRequest;
```

The protobuf representation can be found below:


```

const pb_field_t MatchmakingCoreRequest_fields[11] = {
  {1, (pb_type_t) ((int) PB_HTYPE_OPTIONAL | (int) PB_LTYPE_VARINT),
  offsetof(MatchmakingCoreRequest, PlatformType),
  pb_delta(MatchmakingCoreRequest, has_PlatformType, PlatformType),
  pb_membersize(MatchmakingCoreRequest, PlatformType), 0, 0},

  {2, (pb_type_t) ((int) PB_HTYPE_OPTIONAL | (int) PB_LTYPE_VARINT),
  pb_delta_end(MatchmakingCoreRequest, RequestType, PlatformType),
  pb_delta(MatchmakingCoreRequest, has_RequestType, RequestType),
  pb_membersize(MatchmakingCoreRequest, RequestType), 0, 0},

  {3, (pb_type_t) ((int) PB_HTYPE_OPTIONAL | (int) PB_LTYPE_VARINT),
  pb_delta_end(MatchmakingCoreRequest, RequestNonce, RequestType),
  pb_delta(MatchmakingCoreRequest, has_RequestNonce, RequestNonce),
  pb_membersize(MatchmakingCoreRequest, RequestNonce), 0, 0},

  {4, (pb_type_t) ((int) PB_HTYPE_OPTIONAL | (int) PB_LTYPE_VARINT),
  pb_delta_end(MatchmakingCoreRequest, MatchmakingType, RequestNonce),
  pb_delta(MatchmakingCoreRequest, has_MatchmakingType, MatchmakingType),
  pb_membersize(MatchmakingCoreRequest, MatchmakingType), 0, 0},

  {5, (pb_type_t) ((int) PB_HTYPE_OPTIONAL | (int) PB_LTYPE_VARINT),
  pb_delta_end(MatchmakingCoreRequest, MatchmakingSetId, MatchmakingType),
  pb_delta(MatchmakingCoreRequest, has_MatchmakingSetId, MatchmakingSetId),
  pb_membersize(MatchmakingCoreRequest, MatchmakingSetId), 0, 0},

  {6, (pb_type_t) ((int) PB_HTYPE_OPTIONAL | (int) PB_LTYPE_VARINT),
  pb_delta_end(MatchmakingCoreRequest, ManagedSessionIndex, MatchmakingSetId),
  pb_delta(MatchmakingCoreRequest, has_ManagedSessionIndex, ManagedSessionIndex),
  pb_membersize(MatchmakingCoreRequest, ManagedSessionIndex), 0, 0},
};

```

Note that protobuf descriptions describe the representation of the data on the wire, which makes it language agnostic.

Data Store

Once an API and data protocol are established we explore the other components of the matchmaking server, such as the Data Store. When a client advertises its data, it sends the properties to the server, and the server stores and queries it based on client calls. This data is highly structured. Each advertisement has a finite set of properties with well-defined data types. This means that we have flexibility in choosing a storage solution.

Traditional data technologies such as SQL Server, Oracle, MySQL, are natural choices for storing matchmaking data given the structured nature of advertisement data. There are some tradeoffs for using SQL based solutions. Advantages include the fact that most of the offerings are mature, reliable, and well understood. They also have some

disadvantages such as being expensive and relatively heavyweight and complex. Note that matchmaking data is transient and doesn't need some of the reliability features offered by structured datastores. Early versions of Destiny's matchmaking used SQL to store advertisement data and this was effective in production.

In the last few years new data solutions have emerged under the label of "NoSQL". Many supporting structured and unstructured data. The core idea behind the innovative solutions is to treat data storage as a service and provide different levels of guarantee depending on the needs of the application. Technologies such as Redis and Dynamo have grown in popularity in the web space and offer advantages such as reduced cost and improved scalability. These technologies also have drawbacks such as being new/unproven and being more complex to code against due to supporting unstructured data. Destiny moved from SQL to NoSQL after the first year of production, simplifying the matchmaking datastore complexity considerably.

For the various stages of the server side we have the following:

1. **Update Advertisement:** During this call the server takes the information provided and puts it in the Data Store. If the data is not present, then a new entry is created for the client making the call. This ensures that the system is fault tolerant. If the Data Store goes down it is effectively rebuilt over a short period of time.
2. **Search for Candidates:** During this call the server searches the Data Store for candidates that match the criteria requested by the client. The server has limits in terms of how many results to return, and how many entries to evaluate. The

limits exist as toggles to reduce computation time as searches with multiple criteria can be expensive.

For scalability standard solutions can be explored, such as putting multiple matchmaking servers behind a load balancer and splitting the matchmaking data per region.

CHAPTER 5: MATCHMAKING SKILL

Now that we have explored both the client and server architecture we will explore one of the most critical aspects of a matchmaking system: skill. When considering building a matchmaking system for competitive play the most impactful aspect to consider is how player skill is represented and whether the system is matchmaking players as expected.

Representing skill is one of the most important aspects of any matchmaking system. First person shooters need to balance the need to have good connection quality with the need to provide a fair and balanced experience for players. There are multiple ways to represent player skill, and the ideal is to find matches with low latency, as defined by the game's requirements, while having each team have an identical chance of winning.

It is intuitive to use values such as win count, wins over losses, kill death ratio, etc. Unfortunately those values don't tend to represent player skill well, because they don't take into account the quality of the opponents. It is very different to defeat a brand new player than defeating one of the top veteran players in the game. More sophisticated skill models exist to better represent skill, here are a few:

- **ELO:** ELO was developed by Arpad Elo (Arpad Elo, 1960), a Hungarian-American physics professor, for calculating the relative skill levels of players in head to head games such as chess. This skill system has been widely used in videogames, professional sports, and by multiple chess organizations to track player skill.

- **Glicko:** Glicko was developed by Mark Glickman (Glickman, 1995) as an improvement over ELO. The main difference between ELO and Glicko is that Glicko takes into account the idea of “*rating deviation*” which tries to capture the confidence level of the skill rating. The rating deviation can also take into account skill degradation over time for competition inactivity.
- **Glicko2:** Glicko 2 was also developed by Mark Glickman (Glickman, 2013). Glicko 2 offers improvements over Glicko by introducing the concept of “*rating volatility*” which is used to measure the degree of expected fluctuation in a player’s skill rating. By capturing this idea it is possible to represent how stable or dialed in player skill actually is.
- **TrueSkill:** This system was developed and patented by Microsoft (Microsoft, 2007), and is a Bayesian skill rating system which models uncertainty about player skills. TrueSkill was integrated into Xbox Live and it has been used by several iterations of games in the Halo franchise (Bungie, 2007).

Below you will find the details of both ELO and Glicko, comparison of the simulations, and actual game results.

ELO is well documented, and a full breakdown of how it works can be found at (Arpad Elo, 1960). It is based on the idea that each player in each game is a normally distributed random variable, and that player performance can be estimated statistically based on wins, draws, and losses. Performance isn’t measured absolutely, and changes in rating are dependent on the ratings of the opponent. ELO is modeled after the following variables and equations:

$Ra = \text{Rating of Player A}$

$Rb = \text{Rating of Player B}$

$Ea = \text{Expected Score for Player A}$

$Eb = \text{Expected Score for Player B}$

$Sb = \text{Actual Score for Player B}$

$$Ea + Eb = 1$$

$$Ea = \frac{1}{1 + 10^{\frac{(Rb-Ra)}{400}}}$$

$$Eb = \frac{1}{1 + 10^{\frac{(Ra-Rb)}{400}}}$$

And the equation for updating ratings based on actual scores is the following:

$$R'a = Ra + K(Sa - Ea)$$

$$R'b = Rb + K(Sb - Eb)$$

There are a couple of values in the equations that need some clarification. The first one is the denominator set at 400 in the original equation, and the other one is K used for the rating update equation. The 400 is used to magnify the expected score based on the original rating delta, meaning that every 400 rating difference makes the expected score 10X bigger. We will run through some examples to clarify this. The K factor used on the rating update equation, and it represents the maximum possible adjustment at the end of a game. For Chess the recommend value is between 16 and 32, and depending on the design goals you can use different values of K to define how fast skill can grow.

Glicko's full breakdown can be found at (Glickman, 1995). The foundation is also based on the idea that each player in each game is a normally distributed random

variable, but that on top of the skill rating we also capture the idea of a ratings deviation (RD) which measures the uncertainty in a rating, and it also incorporates the idea of such deviation to be affected by time intervals between competitions. Glicko can model scenarios in which two players have the same skill rating but one hasn't competed in several months causing a mismatch in rating deviations and predicting different outcomes. Glicko is modeled after the following variables and equations:

RD = Rating deviation

t = Number of rating periods since last competition

c = Constant that governs the increase of uncertainty over time

r = Rating

m = Number of opponents in the match

The algorithm to update Glicko skill is the following:

- If the player is unrated then assign default values (For Chess this is 1500 skill rating and 350 RD), or
- If the player is already rated then update the rating deviation (RD)

$$RD = \min(\sqrt{RD_{old}^2 + c^2 t}, 350)$$

- Carry out the following calculation for each player:

$$r' = r + \frac{q}{\frac{1}{RD^2} + \frac{1}{d^2}} \sum_{j=1}^m g(RD_j)(s_j - E(s|r, r_j, RD_j))$$

Expanding further:

$$q = \frac{\ln 10}{400} = 0.0057565$$

$$g(RD) = \frac{1}{\sqrt{1 + \frac{3q^2(RD^2)}{\pi^2}}}$$

$$E(s|r, r_j, RD_j) = \frac{1}{1 + 10^{-\frac{g(RD_j)(r-r_j)}{400}}}$$

$$d^2 = \left(q^2 \sum_{j=1}^m (g(RD_j))^2 E(s|r, r_j, RD_j)(1 - E(s|r, r_j, RD_j)) \right)^{-1}$$

There are a couple of things that are worth noting about Glicko:

1. Changes in rating are not balanced, meaning that Glicko is not a zero sum equation. Rating changes are dictated by the RD of all players involved.
2. A more intuitive way to characterize RD is by reporting a confidence interval.

The original Glicko paper offers the idea of using the interval $(r - 2*RD$ to $r + 2*RD)$ and reporting a 95% confidence over that interval. This allows using language such as “There is a 95% confidence that the player rating is between 200 and 400 skill points” for a player that has a rating of 300 and a RD of 50.

Now that we have a fundamental understanding of both systems we will talk through some examples to see how the system behaves.

Example 1

Two opponents at the same skill level (1500), A beats B

ELO

Player	Initial Rating	Final Rating	Delta
A	1500	1508	+8
B	1500	1492	-8

Glicko (Using wide RD of 350)

Player	Initial Rating	Initial RD	Final Rating	Final RD	Rating Delta	RD Delta
A	1500	350	1662	290	+162	-60
B	1500	350	1337	290	-163	-60

Glicko (Using RD of 200)

Player	Initial Rating	Initial RD	Final Rating	Final RD	Rating Delta	RD Delta
A	1500	200	1578	180	+78	-20
B	1500	200	1421	180	-79	-20

Glicko (Using narrow RD of 50)

Player	Initial Rating	Initial RD	Final Rating	Final RD	Rating Delta	RD Delta
A	1500	50	1506	49.5	6	-0.5
B	1500	50	1493	49.5	-7	-0.5

Glicko (Using different RDs, 350, and 50)

Player	Initial Rating	Initial RD	Final Rating	Final RD	Rating Delta	RD Delta
A	1500	350	1675	248	175	-102
B	1500	50	1495	49.79	-5	-0.21

As you can see in the ELO case the rating delta is 8, but the Glicko delta varies between 6 and 160 depending on the rating deviation.

Example 2

Two opponents at widely different skill levels. A beats B. A has a skill rating of 1500 and B has a skill rating of 500.

ELO

Player	Initial Rating	Final Rating	Delta
A	1500	1500.05	+0.05
B	500	499.95	-0.05

Glicko (Using wide RD of 350)

Player	Initial Rating	Initial RD	Final Rating	Final RD	Rating Delta	RD Delta
A	1500	350	1509.5	343	+9.5	-7
B	500	350	490.5	343	-9.5	-7

Glicko (Using RD of 200)

Player	Initial Rating	Initial RD	Final Rating	Final RD	Rating Delta	RD Delta
A	1501	200	1501.5	199.3	+1.5	-0.7
B	500	200	498.5	199.3	-1.5	-0.7

Glicko (Using narrow RD of 50)

Player	Initial Rating	Initial RD	Final Rating	Final RD	Rating Delta	RD Delta
A	1500	50	1500.048	49.99	0.048	-0.01
B	500	50	499.952	49.99	-0.048	-0.01

Glicko (Using different RDs, 350, and 50)

Player	Initial Rating	Initial RD	Final Rating	Final RD	Rating Delta	RD Delta
A	1500	350	1502.3	347.68	2.3	-2.32
B	500	50	499.79	49.98	-0.21	-0.2

This case is very interesting because we are trying to understand how the systems behave when the outcome of the match is expected and there is a big difference between player ratings. For ELO the rating change is very small, but for Glicko that change is dictated by the rating deviation. With a fairly open rating deviation we can expect the rating to move by ~10 points, but if it is narrow then Glicko ends up behaving like ELO.

Example 3

Two opponents at widely different skill levels. B beats A. It is an upset. A has a skill rating of 1500 and B has a skill rating of 500.

ELO

Player	Initial Rating	Final Rating	Delta
A	1500	1484	-16
B	500	516	+16

Glicko (Using wide RD of 350)

Player	Initial Rating	Initial RD	Final Rating	Final RD	Rating Delta	RD Delta
A	1500	350	1054.5	343.7	-445.5	-6.3
B	500	350	945.5	343.7	+445.5	-6.3

Glicko (Using RD of 200)

Player	Initial Rating	Initial RD	Final Rating	Final RD	Rating Delta	RD Delta
A	1500	200	1308.5	199.3	-191.5	-0.7
B	500	200	691.5	199.3	+191.5	-0.7

Glicko (Using narrow RD of 50)

Player	Initial Rating	Initial RD	Final Rating	Final RD	Rating Delta	RD Delta
A	1500	50	1485.8	49.99	-14.2	-0.01
B	500	50	514.2	49.99	+14.2	-0.01

Glicko (Using different RDs, 350, and 50)

Player	Initial Rating	Initial RD	Final Rating	Final RD	Rating Delta	RD Delta
A	1500	350	815	347.68	-685	-2.32
B	500	50	509	49.98	+9	-0.2

Modeling an upset provides information on how well both ELO and Glicko adjust to unexpected changes. For ELO a significant upset creates a rating delta that is dictated by the choice of K. For Glicko the change in rating varies depending on the rating deviation. In the case of an upset when the RD is open we see wide swings on the rating, but once the RD shrinks then Glicko lowers the potential rating changes because there is a high level of confidence that the players are already at the right level.

ELO vs Glicko Simulation results

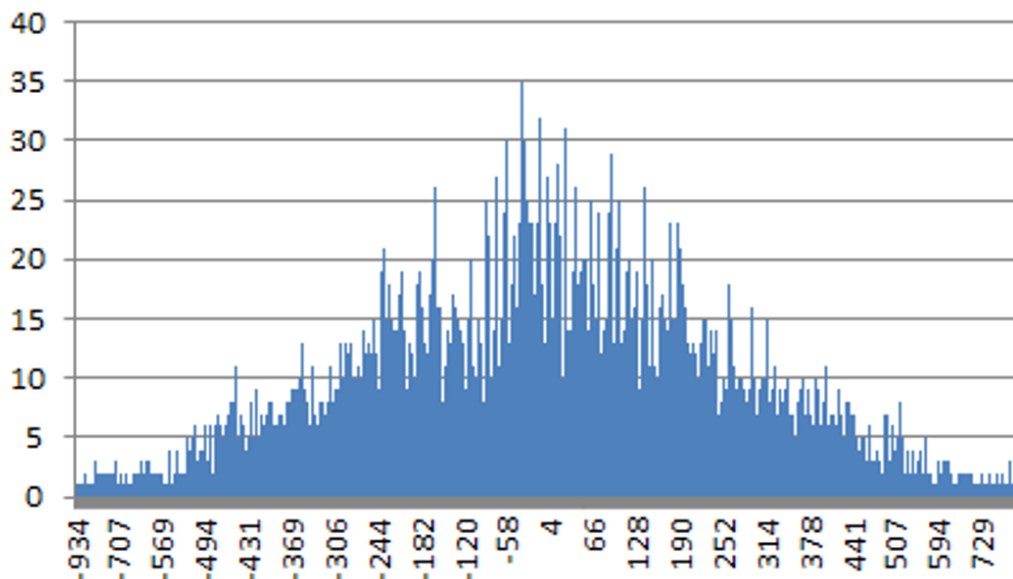
Another way to compare the algorithms is by running simulations and comparing the convergence rate and the final distribution. For this purpose I built a simulator that had a current rating and a real absolute rating for each player. The idea was for the simulator to build matches with random players but at the end to report the outcome of each match based on the absolute real rating. The absolute ratings were assigned using a normal distribution curve.

The simulations ran 8000 iterations, with a population of 1000 players. Each iteration would pick up to 12 players at random and create a match with an outcome. At the end of the simulation I analyzed 3 pieces of data:

1. The rating distribution
2. The delta between the final rating and the absolute rating
3. The historical results for some of the players to analyze convergence

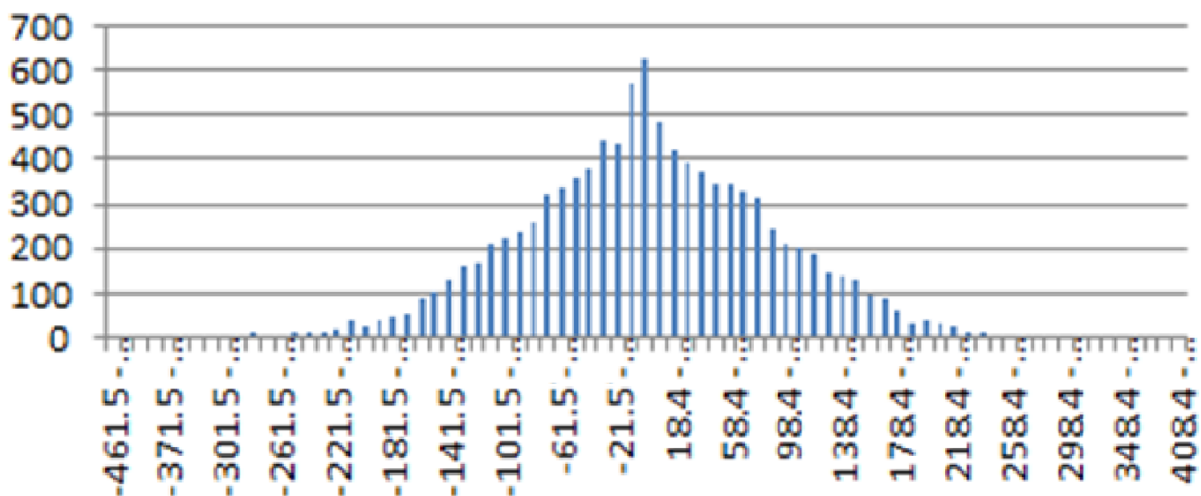
Let's start with ELO's final distribution (Figure 4):

Figure 4 **Skill Distribution**



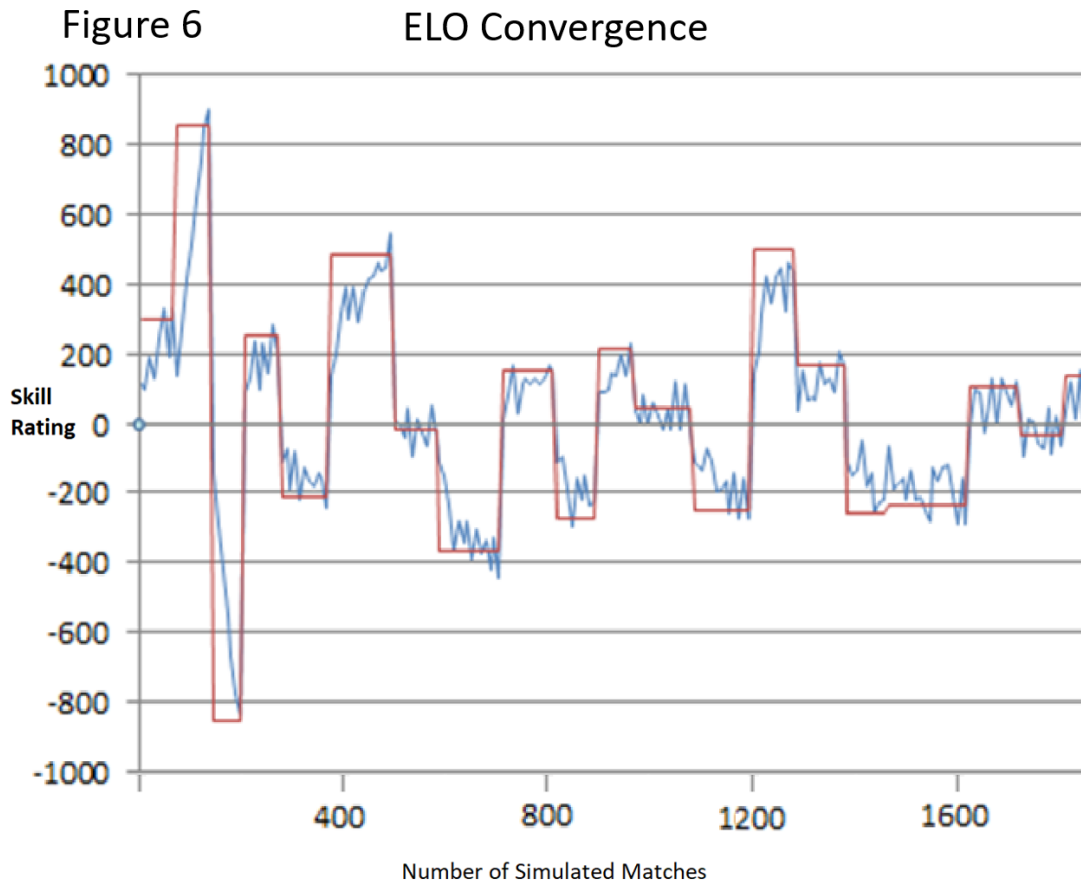
As you can see ELO does a fair job generating a final skill distribution that resembles a normal distribution. This means that at the end of the simulation players ended up roughly close to the expected absolute rating level. The next question we want to answer is finding how close they were. For that we use the rating delta graph (Figure 5):

Figure 5 **Average Delta From Expected**



At the end of the ELO simulation we found that 80% of the players were within 100 rating points for their absolute real rating. On a -1000 to 1000 scale what we get is that most players are within 5% of their absolute rating at the end of the simulation.

Another piece of data that is useful to visualize ELO is convergence. I used a few pieces of historical data for random players and I plotted their progress against the ideal final graph. Results can be found below:

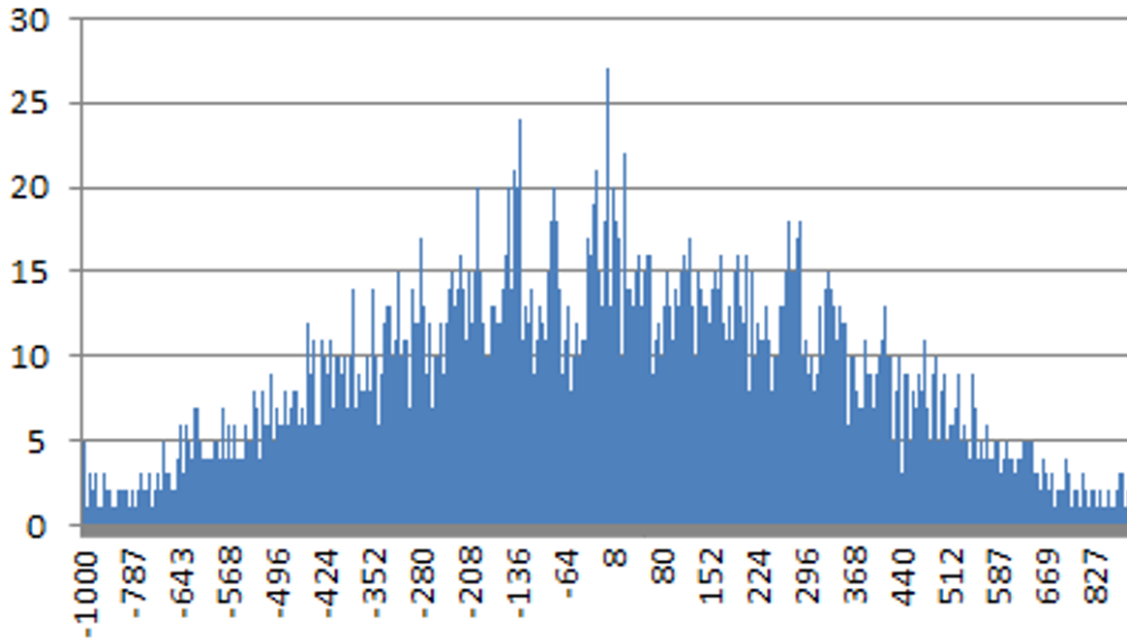


What this graph tells us is that ELO converges, and eventually gets close to the final expected rating, but the rate of convergence is slow. In most simulations convergence could take dozens of games, and different values of K would change the convergence rate. Higher values of K could allow faster convergence but can also result in oscillation about the expected outcome. Remember that K in the ELO equation indicates what the maximum point trade allowed is during the calculation. The graph above uses a K value of 16, which is the default used for Chess.

Now let's look at the equivalent graphs for Glicko. Let's start with the skill distribution graph elow:

Figure 7

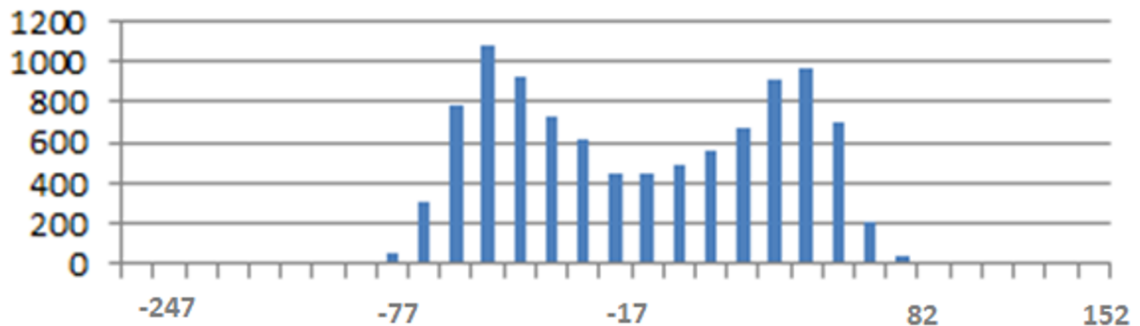
Skill Distribution



Much like ELO, the final distribution resembles a normal distribution. So at this point we believe we have a good starting point for capturing player skill. Let's move on to analyzing the final rating delta at the end of the simulation. The graph below shows the results:

Figure 8

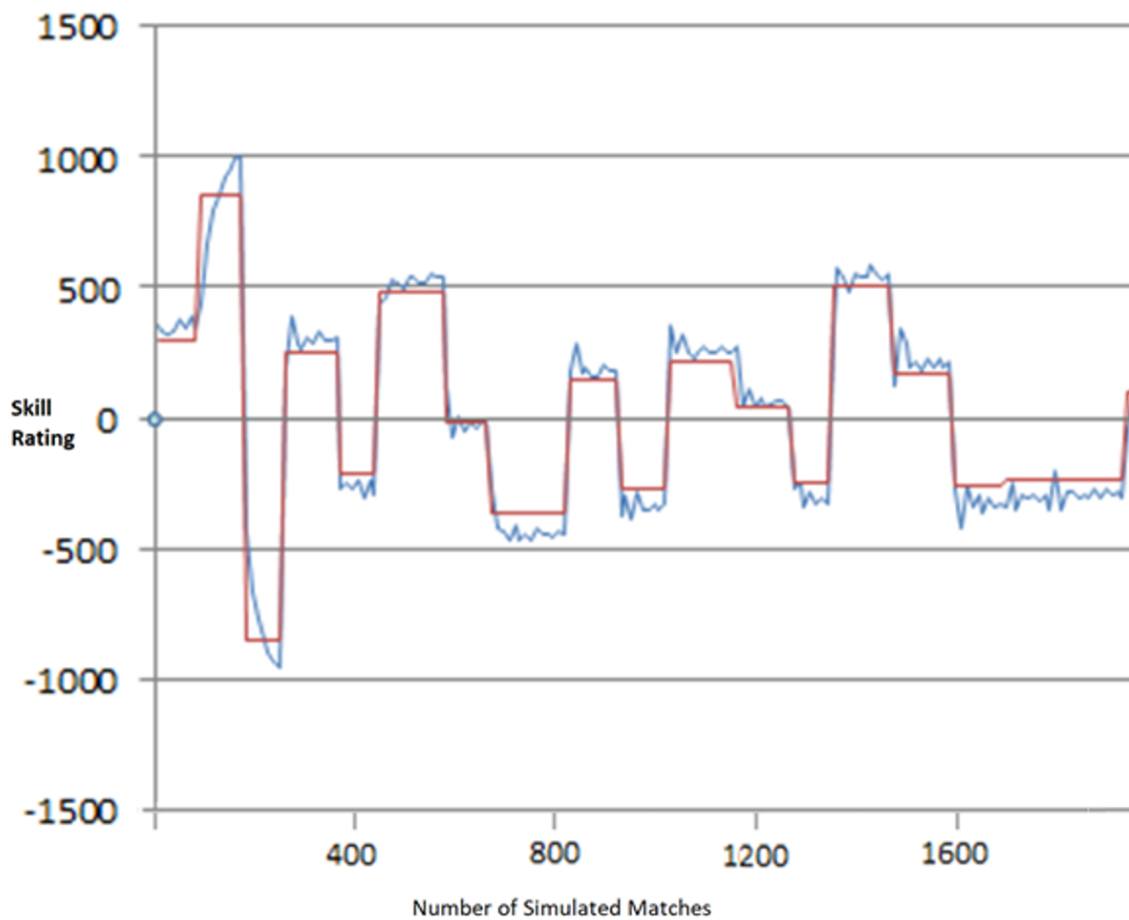
Average Delta From Expected



For Glicko we can find that over 80% of players are within 50 rating points from their absolute real rating. On a -1000 to 1000 scale what this means is that the final rating is within 2.5% of the real rating. That is a considerable improvement over ELO.

The final piece we want to analyze is convergence for Glicko. Below is the graph showing convergence for a few players:

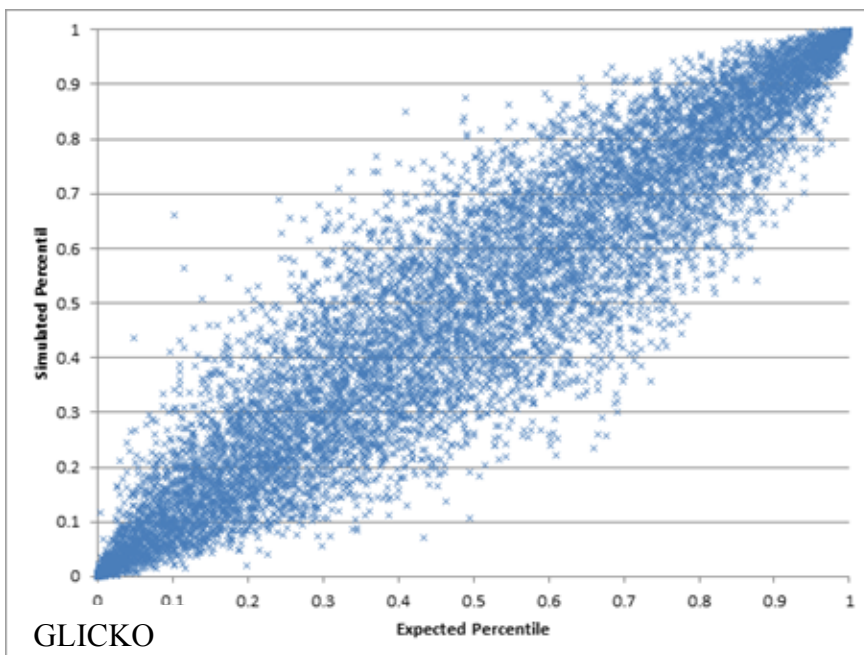
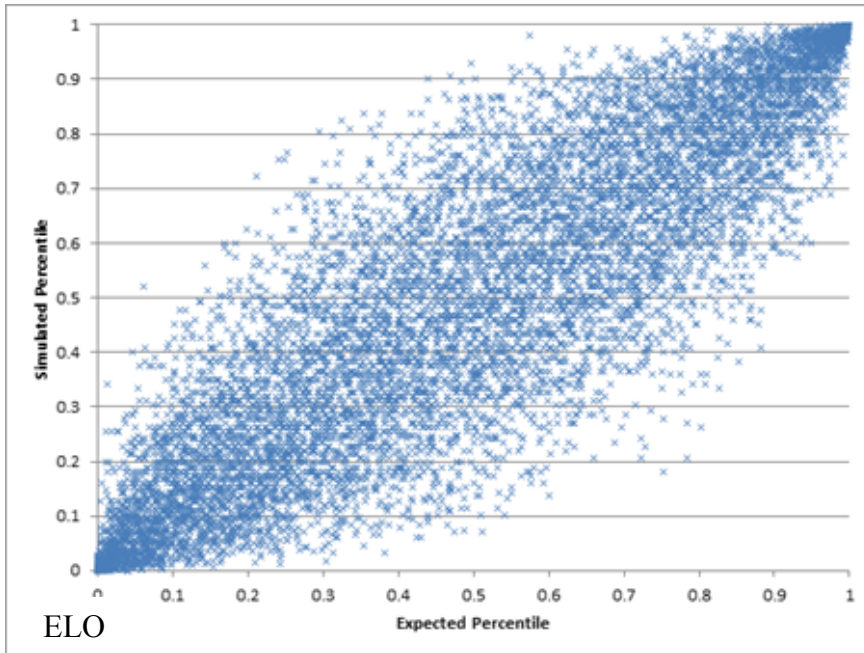
Figure 9 Glicko Convergence



Convergence for Glicko is fairly good. During the simulation most players converged within 10 matches. Compare this to ELO in which players could take dozens of matches to get some convergence. This is a considerable improvement that Glicko has over ELO.

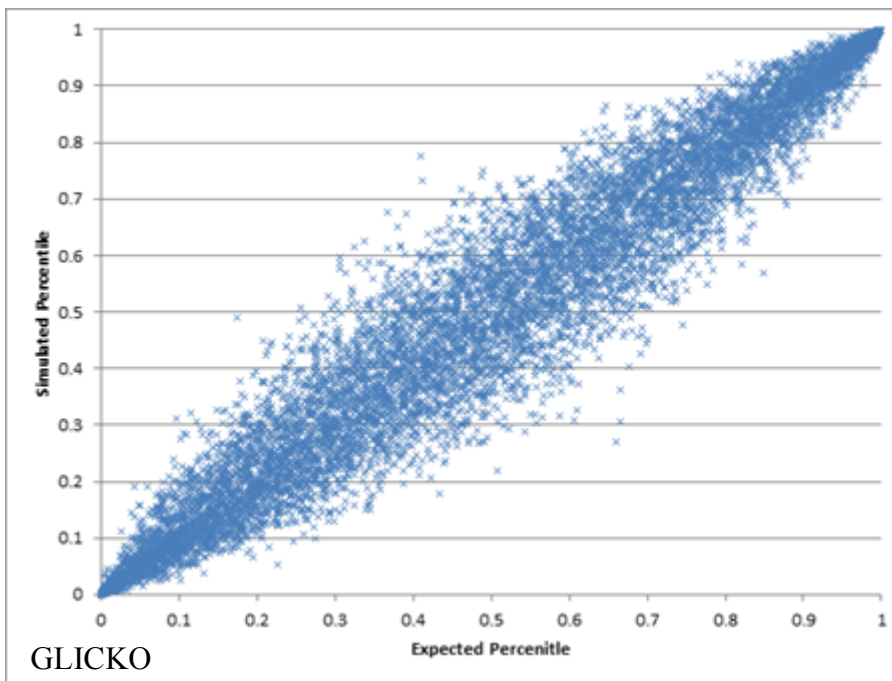
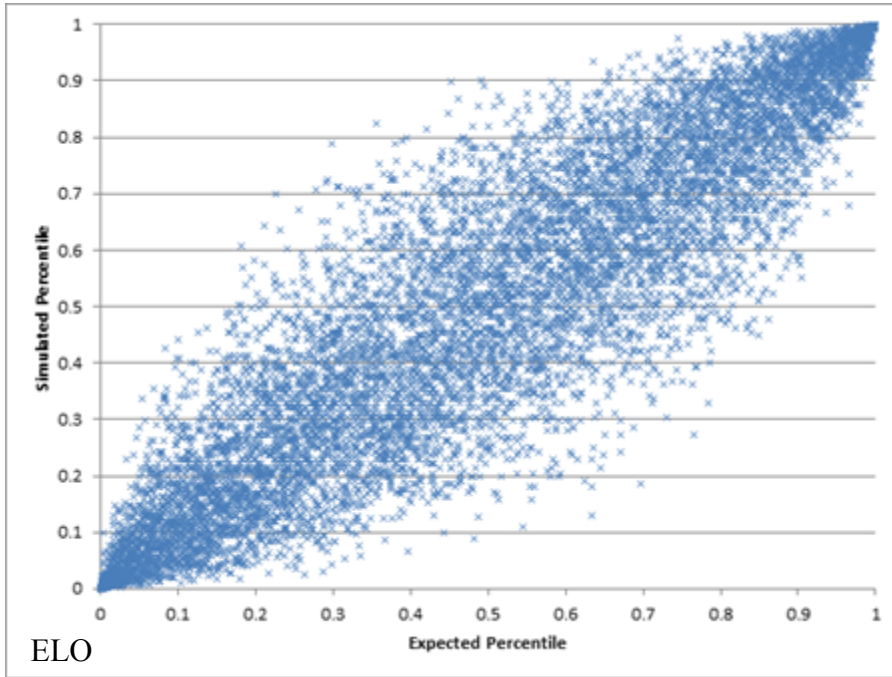
Another way to visualize convergence is to plot simulated result vs expected result. We moved this into percentiles to normalize the results. I ran simulations for 1000 players through multiple simulation loops and the results can be found below for 10, 24, 100, and 200 matches per player.

10 matches per player, ELO vs Glicko



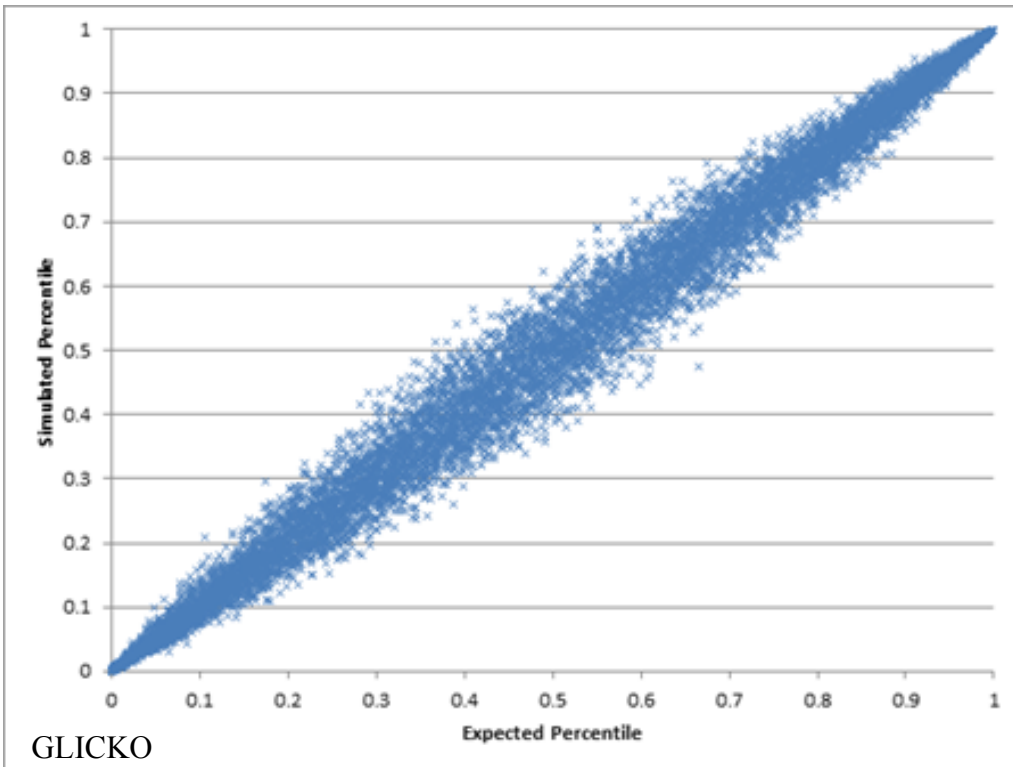
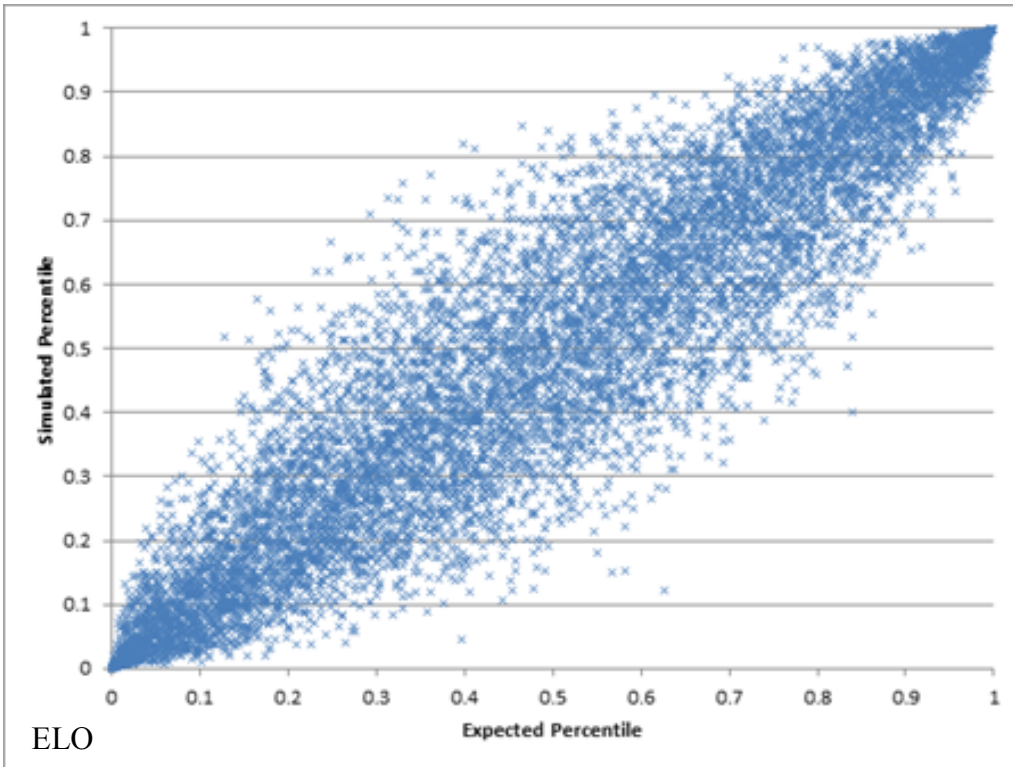
At the end of the 10 matches ELO still has a pretty wide spread, but we can see Glicko having a tighter convergence band.

24 matches per player, ELO vs Glicko



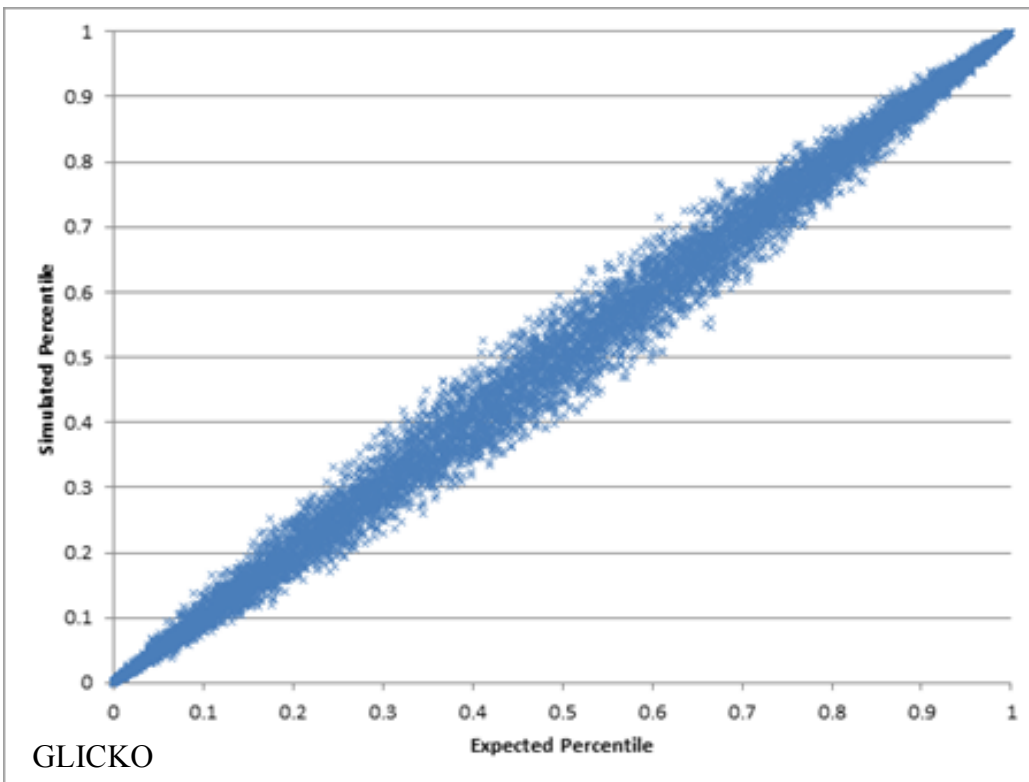
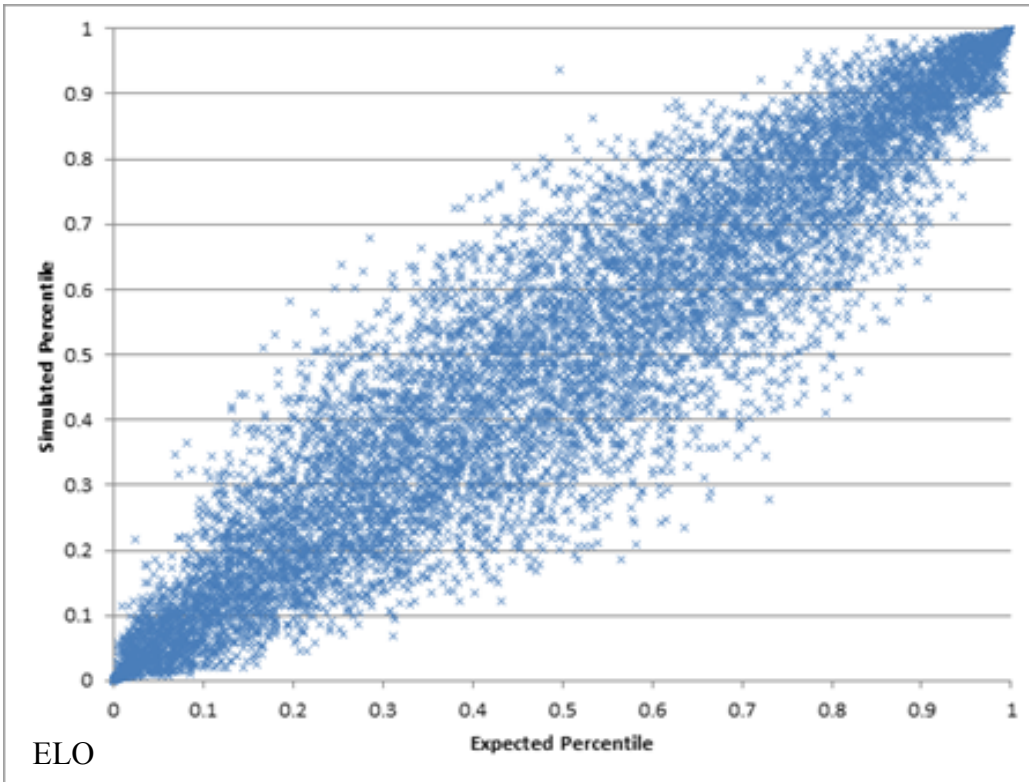
Glicko's convergence is superior at 24 matches.

100 matches per player, ELO vs Glicko



Glicko starts to show high convergence at 100 matches.

200 matches per player, ELO vs Glicko



The results at 200 matches are very interesting. It shows that using completely random match selection for ELO creates several outliers that have not converged. This could be due the random nature of the simulation, and a more targeted match selection process would yield better results for ELO. Glicko holds up well, even under the constraints of the simulator. In reality, when implementing matchmaking for a multiplayer game random matches aren't used. Instead players are matched with other players with similar skill. That would improve convergence for both ELO and Glicko.

CHAPTER 6: FUTURE WORK

Matchmaking is an area that is critical for games and will continue to evolve as games become more connected and more social. There are many ways to keep advancing Matchmaking systems, with the following critical areas that could drive innovations in multiplayer gaming:

1. **Server-Side Matchmaking:** The system presented in this work keeps most of the logic of the Matchmaking system running on the client. This was an artifact of the expertise we built at Bungie while working on the early versions of the system. A simple evolution would be to move all the Matchmaking logic to the server side. The state machine necessary to understand the roles of clients and find the optimal global groupings can be moved to the server with tangible benefits such as: maintaining statistics about search times for all players and remembering previous groupings to avoid repeating them. Once the data and logic lives on the Server there are many extensions that can be explored to improve the Matchmaking system.
2. **Rankings and Skill systems that are satisfaction aware:** One of the biggest challenges for multiplayer games is understanding when players are satisfied with their matches. Skill systems do a fantastic job trying to capture player ability, but players have different expectations during their game sessions. Sometimes they want to play highly competitive matches, and other times they just want to have an enjoyable time. This has caused a lot of friction in multiplayer games and players complain about the idea of having “sweaty” or high-stakes matches all the time. Research into building ranking and skill systems that focus on player satisfaction while maintaining a healthy ecosystem could tackle this problem. Systems can detect

when players are frustrated or losing too many matches in a row and adjust the criteria for the matches accordingly. Feedback loops can also be created to have players provide scores for their matches and use that information to improve matches in the future.

3. **Machine Learning:** With the latest advancements in machine learning it is worth looking at Matchmaking from a different angle. Matchmaking systems provide substantial amounts of data about player matches, quality, and actual outcomes vs desired outcomes. At the simplest level Machine Learning can be used to identify properties that are important for Matchmaking by comparing actual outcomes and desired outcomes, and then using that data to extract new properties that need to be evaluated in the Matchmaking algorithm, fine tuning the match evaluation criteria. More advanced Machine Learning techniques can be leveraged to better understand the contextual nature (competitive vs casual) of the experience and define new evaluation criteria.

BIBLIOGRAPHY

Bungie. "Halo 2." Microsoft, 2004, Redmond, WA.

Bungie. "Halo 3." Microsoft, 2007, Redmond, WA.

Bungie. "Destiny." Activision, 2014, Bellevue, WA.

Bungie. "Destiny: The Taken King." Activision, 2015, Bellevue, WA.

Coleman, S. "The TCP/IP Internet DOOM FAQ." Gamers, 1995,

<http://www.gamers.org/dhs/helpdocs/inetdoom.html>.

Demonware. "About Us." 2016, <https://www.demonware.net/>.

Arpard Elo. "ELO Rating system." 1960,

https://en.wikipedia.org/wiki/Elo_rating_system.

Glickman. "The Glicko system." 1995, <http://www.glicko.net/glicko/glicko.pdf>.

Glickman. "Example of the Glicko-2 system." 2013,

<http://www.glicko.net/glicko/glicko2.pdf>.

International Telecommunication Union (ITU). "Internet Live Stats." Feb. 1, 2016,

<http://www.internetlivestats.com/internet-users/>.

id Software. "Doom." GT Interactive, 1993, Richardson, TX.

id Software. "Quake." GT Interactive, 1996, Richardson, TX.

id Software. "Network setup instructions." Doom, 1995,

<http://www.classicdoom.com/doominfo.htm>.

Microsoft. "TrueSkill." 2007, <http://research.microsoft.com/en-us/projects/trueskill/>.