# Match Making

Several existing services were looked at as well as a few open sourced frameworks. This was not an exhaustive analysis, just a quick look at existing frameworks. I personally have some experience with the Playfab matchmaking service, but no direct experience with the other services. The design of the simple service was based in part on some of the design patterns observed in other services, although their robust nature was not as applicable to this problem.

## 3rd party frameworks:

Amazon FlexMatch as part of Gamelift – Bit.Fry is already using Gamelift, so there should be some synergies here. Like other broad solutions, though it is initially more robust than probably needed for basic matchmaking.

Microsoft Playfab – Used in many large scale projects (Roblox), but is not very optimized for two player direct matches with low-latency timing, however it does scale very well. The overall structure/design is very similar to the Gamelift’s.

GameSparks – used by many large scale games it is now owned by Amazon, but is maintained as a separate product, although it is highly integrated in to the AWS framework. General reports from the field indicate its gotten too expensive for indi development (at least).

## OpenSource frameworks:

Open-Match – This is a relatively new solution available via Google and partnered with Unity among other. Its written in GO, and seems highly scalable, but is integrated heavily with Google cloud services. Due to the open source nature it does seem more easily malleable than Amazon or Microsoft, but has particular language requirements. GO is an “easy” language. Although robust, the framework is also in development, its primary advantage is the open/complete source (much like the advantage that Unreal can have over Unity).

## Roll-you-own vs. Matchmaking services:

A basic 2 player, 1 dimensional matchmaking service is relatively easy to build and deploy even at scale, however as the matching become more complex, either with multiple dimensions and multiple players or teams, the development time increases exponentially. Although basic functionality will be fairly easy, the number of edge cases, error handling, and basic QA needed explodes as more dimensions are added. If there are no plans to move beyond 1 v 1 matchmaking in the near term (6 months to a year) it seems worth building a simple service. However, building and maintaining a matchmaking service will rapidly eat up significant costs. It would take a more in-depth analysis of the costs of running the services to really answer this question. If it costs <10k per year to run all the services on, say, GameLift, then that’s clearly a better solution than rolling our own, however I have not done the full business case across all the services as it’s a little tricky to parse out exactly what the costs would be without bandwidth usage and playerbase numbers.

## The simple solution:

This solution is built around a Player Power Level. Its one dimensional, i.e. players a matched solely on their PPL. This could be basic level based, or something Elo. Calculating the player’s power level is beyond the scope of this project, its assumed that there is a way to assign or calculate the player’s power level.

The goal is rapid matching of 2 players of similar power levels. Its assumed that once matched players will be competing using a peer to peer connection, although a client server match system would not significantly change the algorithms.

The server will maintain a series of queues that each cover a range of levels relatively tightly. This could go as far as a separate Queue for each power level, but initial should start with a broader range and have queue’s tighten over time (note: adding queue’s and tightening there ranges was not considered extensively for this test). For extremely large populations queue’s could eventually be divided across regions or servers or along other lines to separate populations. There may also be a need for non uniform ranges, i.e. if many players are heavily grouped in the ranges 1-10 we may desire 10 queues for those players, but the 11th queue may cover range 11-15, etc… For simplicity purposes here, I’m assuming a uniform queue size.

Note I use the term player and client relatively interchangeably here; i.e. I’m assuming there’s a one-to-one match with each player having a single client (mobile device, pc, etc…). Also the term Queue may be a mis-nomer as there technically should only ever be one player in the waiting area with the current implementation, but at some point it will likely be switched to a queue to support omin-directional matching, or more variant match styles.

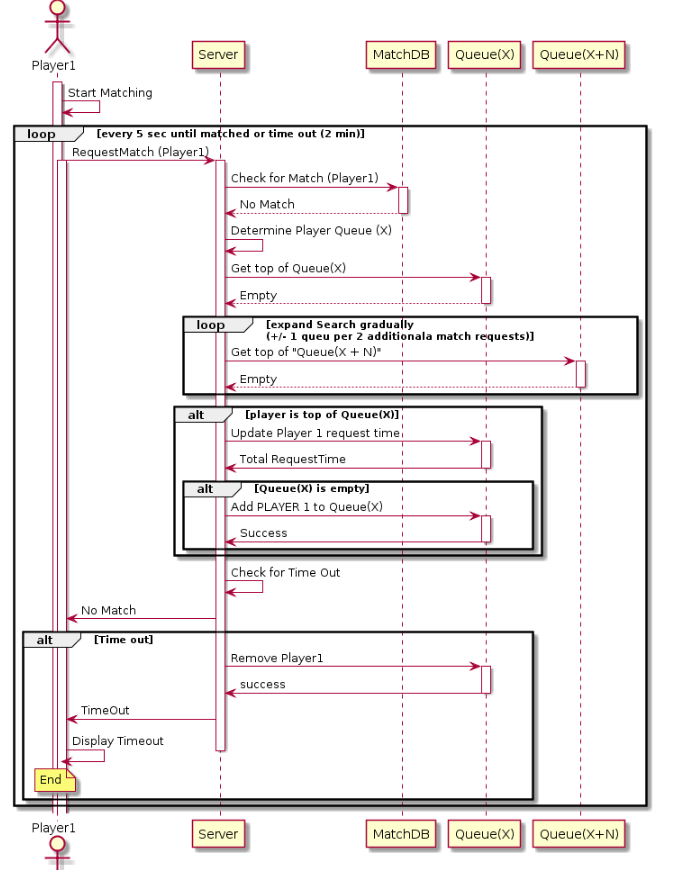
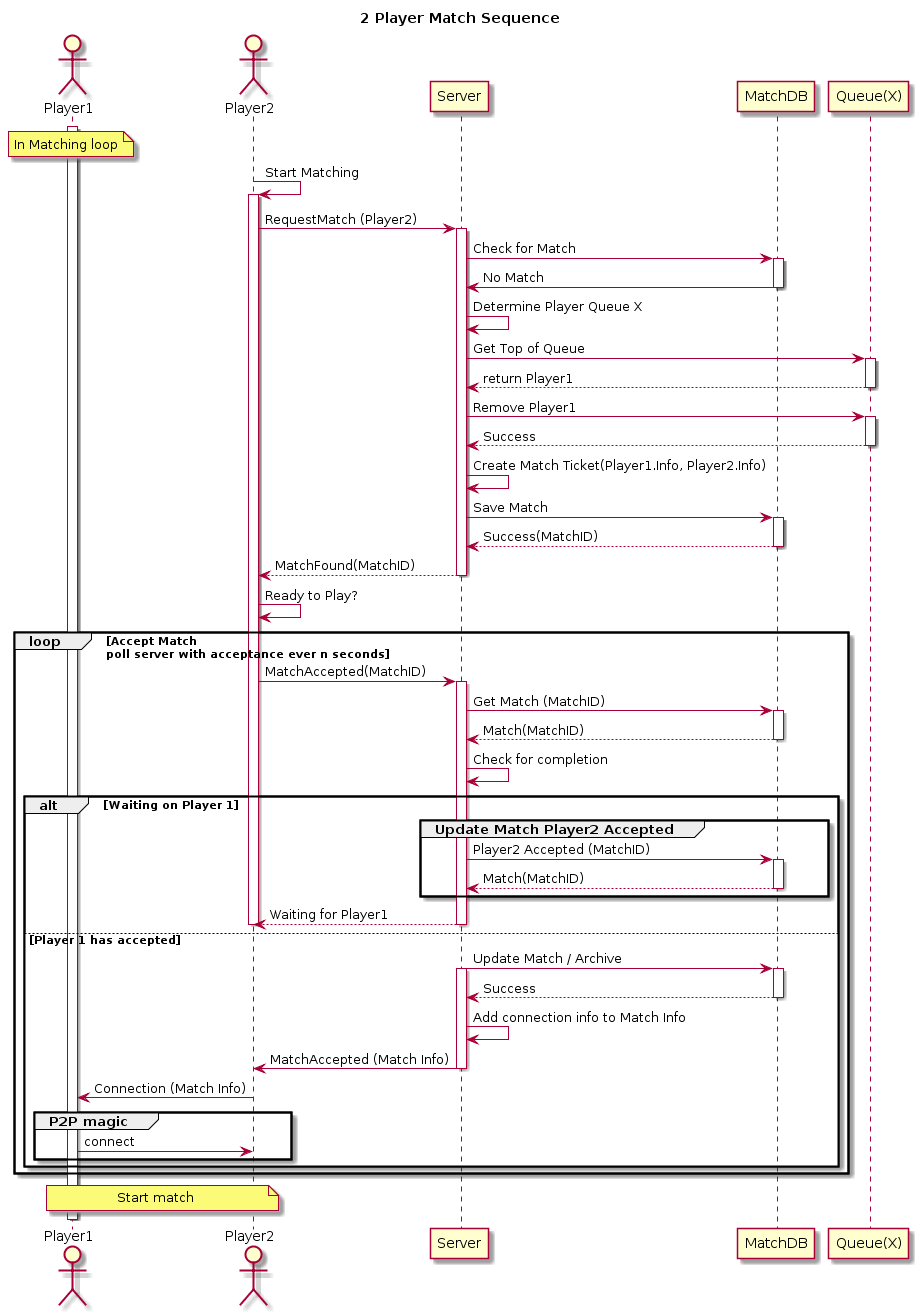


Figure : Player 1 starts the flow with a match request

Once a player has requested match, the server checks for a match in the match database, and then if none is found the server checks the appropriate queue based on their power level. If there is someone in the queue, the server creates a match and sends the matching info to the player and removes the matched player from the queue. If there is no match the player is added to the queue. When a player makes a request for a match, if they’re in the queue, then the timestamp on their request is updated. The match request is a polling request and will fire every n seconds (exact time tbd) from the player to the server. As the player’s time increases since they requested the match we will start poling adjacent queues to find matches, gradually increasing the power level range that the player can be matched in. Eventually if no match is found the player is informed and removed from the queue. Note: the expanding search is not reflective, it does not require both players to be seeking outside their bands - it’s possible for a new match request to be “grabbed” by an older match request that is seeking outside its bounds. The older request’s need for a match overrides the new matches band, i.e. if player 1 request a match in band 5, and later player 2 requests a match in band 1, if enough time has passed since player 1’s initial request, on their next request they might grab player 2 for a match, although player 2’s search has not expanded to the breadth of player one’s search.



When a player is matched, the timestamp on the matching player is checked and it if is stale (over some threshold), then that player is discarded from the queue and the matching process continues -- this handles gracefully the cases where a player requests a match and then disconnects, or puts down their device or otherwise interrupts their polling before they are cleared from the queue. Otherwise when a player is matched, the matching player is removed from the queue and a match is created and added to a separate match database that contains all the information on the two players. This match info is then passed to the player for acceptance.

Once a match is found the client switches from polling for a match to polling match acceptance (or possibly denial, but for now its assumed that if the player matches with another player they accept the match). Both players involved in a match must accept the match. The first player to accept will be assigned as the host of the match, and their info added to the match item in the match database, and the second player will use that information to connect to the first player for the peer to peer engagement. Once the second player has accepted the match, the match will be removed from the match database and in to a match results database that can then be updated based on the actual match results (not in the scope of this).

For the purposes of this exercise I’ve build a simple node.js server that provides exposes very simple services for the matchmaking. I’m assuming a simple range of 5 levels per queue for the player’s power level, so the queue number is just PPL/5. Were I to expand this it queues of multiple ranges I would maintain a look-up table / DB to match power level to queue.

To expand beyond the simple 1v1 approach at a naïve level we continue to use the power queues for the initial match, but we add a another DB for unfilled matches to fill it with additional players, and we sort the DB by PPL or divide it in to queues itself to speed lookups and matching. Several other factors were not really considered in this such as security, or some basic error checking such as duplicate player id’s, stale matches, etc… The system could easily support multiple games with the addition of a gameID field, then the server would either maintain separate queues for each game, or just request the oldest player with a given gameID from the queue. Matches would also be requested based on game ID.