A Survey on Peer-to-peer Network Architectures in Video Games

Arrian Chi
alienchi@ucla.edu

***Abstract*—Traditionally, game developers develop online multiplayer features by leveraging server-client architectures due to its simplicity, security, and synchronization benefits. However, research conducted in the last 2 decades has shown that implementing networked features with peer-to-peer architectures can theoretically incur great benefits, including robustness, lower latency, and higher scalability. In this paper, we discuss the primary problems in gaming, the limitations of traditional server-client architectures, various techniques to realize the peer-to-peer solutions, and finally, the use of peer-to-peer in commercial games.**

I. Introduction

The video game industry is one of the biggest entertainment industries in the world. In 2022, the global gaming industry generated an estimated $184.4B in revenue, compared to the music industry's $26.2B and the movie industry's $26B [4]. The industry is only going to grow larger, with a projected revenue of $583.69 billion in 2030. Online games accounted for the largest revenue share of around 44% [1]. With so much money involved, it is crucial that the development, deployment, and maintenance of games is executed as optimally as possible.

When developing online multiplayer games, the number one priority is a playable gameplay experience. Investigations have shown that game-related network factors have a significant influence on a player's quality of experience [14]. Network issues frequently lead to players leaving the game, so if developers want to increase the quality of their game, they must consider addressing the underlying network issues. On a high level perspective, these may include:

- Connectivity: Players need to have a means of interacting with each other across the internet. The game needs to match players to each other, implying the need to aggregate player data somewhere for group matching. This is usually called matchmaking, a process that takes in a list of players, and outputs groups created from this list.
- Scalability: As a game's player-base grows, the resources required to support them increases. Developers must formulate ways to reduce overhead in hardware and communications while maintaining a playable gameplay experience. There are multiple ways to do this, most commonly by buying new hardware to run multiple instances of the server program in different geographic locations of the world. However, research shows that this method starts to incur diminishing returns.
- Latency: The logic for the game's main loop is executed several times a second. This includes sending/receiving data to servers/clients, processing the responses, and rendering the output on the screen. To provide smooth gameplay, the game loop must keep up with the screen's frame rate, thus latency of game state intra/intercommunication must be minimized. Usually called "lag" in gaming communities, latency issues can arise from a multitude of areas, such as poor hardware performance, connectivity issues, or unoptimized code. Solutions encompass improving these mentioned areas or completely hiding latency to provide the illusion of real-time gameplay.
- Synchronization: Because of latency issues, packets sent between players may be delayed or corrupted. The game needs to reason about these packets and order them in chronological order to ensure fairness and consistency in gameplay. Due to time constraints, I will not be delving into this issue, but it is important to note that synchronization is a major issue in networked games.
- Security: Online games must have a mechanism of preventing/punishing cheaters. Most developers focus on the former by using modern security techniques, but the latter is also effective at deterring players from hacking their clients. This issue is very important in competitive games, since players with unfair advantages ruin the gameplay experience of other players. Additionally, good security is important for games with microtransactions to ensure that the game economy itself does not implode and hurt the game's financial model [18].

On the lower level perspective concerning implementation, online multiplayer game developers must choose a networking architecture to serve as the networking mechanism for their game. Most commercial games leverage the server-client architecture to connect their players. To put simply, all players run clients which connect to a server. The server serves as a central hub for all input messages, synchronizing the game state and sending this data back to the clients. The client handles the dispatch of keyboard/controller input from players and renders the state output from servers. The server-client architecture is appealing to developers because of its synchronization property as well as its ease of implementation and enhanced security by design (game state logic is protected by server, which is owned by developers) [14].

But there are some major flaws in server-client architectures. First, servers act as single points of failures. If the server hosting a game shuts down for whatever reason, then the game is also shutdown. Players are unable to play the game during this time, potentially hindering the gameplay experience. Additionally, servers can serve as a bottleneck, increasing latency. If the computational load suddenly increases and the hardware cannot process within the projected timespan, the server imposes major consequences on the latency. Finally, servers are expensive to maintain compared to a client. Game developers need to ensure servers are up-to-date, secure, etc. If the playerbase is considerably large, then more machines need to purchased and maintained, incurring more costs on the developers [5].

Peer-to-peer (P2P) architectures directly solve the problems server-client architectures have. By design, peer-to-peer does not rely on a central communication point to redirect messages, thus the possibility of having single points of failures and bottlenecks are dependent on the average performance of the peers. The cost of maintaining P2P amounts to the cost of maintaining the client, which doesn't require the developer to run a server all the time.

P2P also has pitfalls. Data synchronization becomes a daunting task because the game state has no ground truth to subscribe to. Security also becomes harder since the attacker has direct access to the game state (potentially spoofing it during networked gameplay, for instance). Finally, peer-to-peer is overall more complicated to implement and is not a default option when developing networked features in games.

Research has been conducted to try to mitigate these problems to make peer-to-peer more appealing to the average indie developer [3]. In the following sections, I will present the issues of connectivity, scalability, and latency [1] issues in games and present peer-to-peer solutions for these issues. Towards the end, I will discuss the feasibility of them in commercial games and why this is a question of design.

II. P2P SOLUTIONS TO GAMING

In this section, I will describe intuitions behind some peer-to-peer solutions to the aforementioned problems. Each section shall begin with a brief explanation of the server-client solution, followed by a description of some peer-to-peer solution(s). The solutions are based on research conducted in the last 2 decades, and are not exhaustive. The solutions are also not mutually exclusive, and may be combined to form a more robust solution.

A. Connectivity

Matchmaking in games using the server-client architecture is a simple task. Because the players (clients) all connect to a central server, the server has access to a list of players and can group them based on some matchmaking algorithm implemented in the server code. The grouped clients then receive an announcement that the match has been found, thus beginning their match sessions.

[1]I left out security because I am not too familiar with the subject at hand and time was limited. In all honesty, the topic itself is very broad.

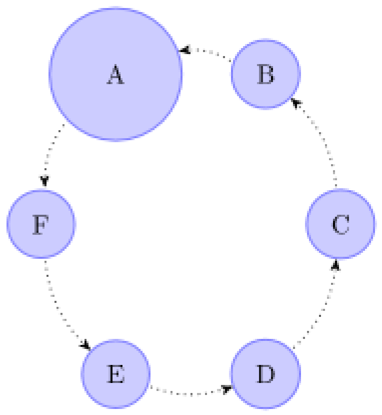

Fig. 1: An example ring structure, where node A is the coordinator and A watches F, which watches E, ..., and B watches A (from [5])

The problem is more complicated in peer-to-peer architectures. Because there is no central server to aggregate player data, each match-seeking peer needs to make themselves known to the other match-seeking peers. Another problem is where to run our matchmaking algorithm (since this must run as long as the game is serviced). There is no central server to run the matchmaking process, yet we need to ensure the algorithm is running at all times.

Before moving forward, it is important to note that I assume that there exists a mechanism for establishing the peer-to-peer network. One common solution is to have a centralized system that creates these connections as in [2]. Players connect to a central server, which sends them the address of peers to connect to. Although this is not a peer-to-peer solution, it is a practical solution, used by many games today. For the rest of this paper, we assume that the peer-to-peer network is established as this is a topic that applies to peer-to-peer networks in general.

Boroń et al. proposes the Self-Aid network, which organizes peers into ring structures(each having a pointer to another, with the last looping back to the first) lead by a coordinator node that publishes their existence and current peers to a distributed hash table (DHT). A DHT is a data structure that maps keys to values (like a hash table), but these key value pairs may exist across multiple networked machines. In our case, the key for matchmaking would be the (unique ID of the) matchmaking algorithm and the value would be the location of the coordinator node and its rings. Players who seek for matches would query the DHT, find the location of the ring, and send a request to the coordinator node. The ring runs the matchmaking algorithm and sends the peers to connect to back to the client. The client then directly connects to the returned peers, starting a match session [5].

Their solution solves both the problem of player request aggregation and the problem of matchmaking process host. The ring of a particular service algorithm is unique, thus all players who request to be matched by a matchmaking algorithm in a period of time will be directed to the same ring. The ring is also responsible for running the matchmaking algorithm, and ensuring that coordinators are replaced in case of failures [5].

In addition to regulating the network, the coordinator also manages the load of the ring. The coordinator recruits new nodes to the ring when the load is high and disconnects nodes when the load is low. Nodes are connected in a first-in-first-out fashion to ensure the coordinator is the oldest node. This uses the intuition that old nodes are less susceptible to failure and in the event that it does fail, the coordinator is the next best candidate [5].

For failure management, the system considers three generic cases: when a normal node fails, when a new node that is about to be recruited fails, and when the coordinator fails. When a normal node fails, the coordinator is notified by the failure's watching node and the ring is reconstructed with by connecting the failure's watching node to the failure's watched node. When the new node fails, the ring disconnects and reverts to the state before the failed recruitment. Finally, when the coordinator fails, the watching node becomes the new coordinator and special cases like if the coordinator was in the midst of recruiting are handled. This robust process of enforcing the ring's structural integrity ensures that the published announcement on the DHT is always accurate and up to date [5].

B. Scalability

A game and the hardware hosting them must be able to accommodate for the growth of the playerbase. That is, the developers must account for the possible costs that arise from having more players. Costs may come from the computational overhead from the increased amount of data to process, the communication overhead from the larger volume of messages being sent/received, the cost of hardware required to alleviate the aforementioned overhead, etc. A game is scalable if it can handle the increased load without incurring a significant increase in costs.

Server-client architectures support scalability as much as the server hardware can support it. To explain, as the playerbase grows, more clients connect to the same server, increasing the load on the server. To alleviate this, developers may optimize their server code to handle more clients. This may include reducing messages sent between servers and clients or optimizing calculations made in the game loop with acceleration data structures. But more often than not, developers just buy more machines and run multiple instances of the server program [3]. This is called a server cluster, and is the most common solution to scalability in server-client architectures [12].

On the other hand, peer-to-peer architectures support scalability as much as the peer hardware can support it. In this case, the number of peers connected to the network increases as the playerbase grows. However, there is no need to buy servers for scaling purposes because all computation is done on the peers. This is a major advantage of peer-to-peer architectures, as there is no direct increased cost for the developer. One must note that this just means the responsibility for having adequate hardware is handed to the players. Players must have hardware adequate to run the game and the networked features, so the development costs may increase to make sure the game is playable on a wide range of hardware [3].

Another architecture worth mentioning that may alleviate the problem of poor performant player hardware in general is cloud architecture [14]. In this architecture, clients still connect to the server, but the game is hosted on a remote game client. The client's only purpose is to upload input to the remote game client, which runs the game loop and communicates with the server. The state is then streamed and rendered on the client's screen. This architecture does come with the same bottleneck and scalability problems as server-client architectures, but the problem of hardware is passed onto the cloud service provider. This is a viable solution for indie developers who cannot afford to run server clusters, but can afford cloud services.

Coming back to peer-to-peer architectures, the problem of scalability comes in different forms. As seen previously with Self-Aid, the matchmaking algorithm is executed by a ring of peers. The load(number of clients requesting matchmaking concurrently) is balanced by the coordinator, who recruits/disconnects nodes as the load increases/decreases[5]. In this case, the peers share hardware resources to resolve the computational overhead. Another form of overhead is communication overhead. This form is best illustrated within the realm of Massively Multiplayer Online Games (MMOGs).

MMOGs are games that have an astronomical playerbase. These games usually involve the players navigating a networked virtual environment (NVE) and interacting with other players. NVEs are virtual worlds that contain and keep track of all player and object (NPCs, items, etc.) activities, and as the name implies, is hosted on multiple machines due to its size and huge number of events to process[9]. Networked virtual environments highlight the scalability problems fairly well, but we will focus on the communication overhead problem here mostly.

On server-client architectures, server clusters may handle load balancing events in different schemes. Some may replicate the world on multiple servers, some may split the objects evenly among servers, and some may allocate servers to each host a different part of the world. The last is among the most relevant approach to peer-to-peer architectures, since it is not feasible for all peers to keep track of the entire game world [9]. With zoning, each peer/server manages all the objects and players within its zone, and communicates with other peers/servers when objects/players move between zones. The latter is where the communication overhead problem comes in. When picking a partitioning scheme, we should try finding one that minimizes the number of peers each peer must talk to, so that the number of messages sent/received across the network is minimized [19].

Various partitioning schemes exist to divide the world among peers. To mention some tessellations, we may divide the space with square grids [11], hexagonal grids [9], and even

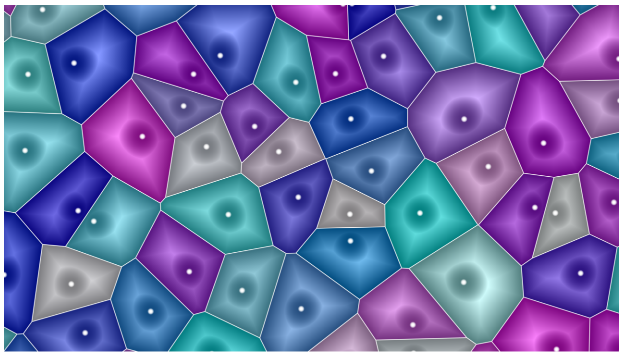
Fig. 2: A Voronoi diagram, every point in each region is closest to the region's site indicated with a white dot

triangular grids [8]. We may also have dynamic/static partitions, whether boundaries are changed at every time step. Hu et al. proposes a Voronoi partitioning scheme, which divides the world into regions centered around players(sites). Every point in a region if and only if it is closest to the region's site and the boundaries are adjusted if the site's position changes [10]. This scheme limits the number of connected peers to those within the peer's area of interest, decreasing communication overhead. In other words, the Voronoi partitioning scheme allows for dynamic interest management, unlike static meshes which require all peers to keep track of the same number of regions.

With any partitioning, each peer is only concerned with events that occur within its area of interest(AOI). We may define the visibility polygon of a peer to contain all peers that are concerned with events that happen in the peer's region and are sent updates when needed. At every time step, the boundaries are modified to match the current positions of the sites. Then the peers connect with peers that have entered their AOI and disconnect from those that have exited. Finally, the peers pass responsibility for an object if the object is no longer in their region [6].

Although Voronoi partitioning alleviates the scalability problem, but there are some pitfalls when crowding occurs. In crowding, assuming the AOI is constant and same for each peer, the Voronoi regions become smaller, and since more regions are within a peer's AOI, each peer keeps track of more peers, increasing communication overhead. There are two methods mentioned that may alleviate this load. One is to aggregate neighboring regions into a single region and designate that peer as a super-peer. The peers within this aggregate region connect to the super-peer for relevant updates in their AOI [9]. However, this assumes that the super-peer has the adequate hardware resources for processing more regions (the super-peer acts like a server connecting all the client-peers). Another method is to have peers dynamically adjust their AOI. In this method, when the number of peers in a peer's AOI exceeds a threshold, the AOI shrinks to limit the number of connected peers, limiting the amount of messages that can be transmitted/received [10]. One caveat is that when disconnecting from peers (due to shrinking), a peer must ensure that the potential disconnectee's AOI does not cover the region, even if the disconnectee is not in the peer's AOI. This is to ensure there exists mutual awareness and that objects are not lost in the networked virtual environment [10].

All this time we have neglected the problem of data management with respect to objects. In the previous sections, we've assumed that the objects within a region are managed by the region's peer. Because region boundaries are determined by peer locations, object managers change as peers move[6].

Though logically sound, it is not the best solution. For instance, if boundaries are changing rapidly due to rapid player movement, then object managers are being changed frequently. The problem is magnified if an area is dense with objects. Because switching object managers implies that memory is being deallocated on the previous peer and allocated on the new owning peer, frequent object manager switching leads to frequent memory operations, which is a major bottleneck in the game loop.

Buyukkaya et al. introduces Vorogame, a hybrid P2P architecture that aims to fix this problem. They separate the object management with the boundary management by having 2 overlays, a structured overlay (the Voronoi representation) and an unstructured overlay (a distributed hash table). Every object thus have 2 responsible peers, a Voronoi responsible peer (VRP) that watches the object for changes and, a DHT responsible peer (DRP) that holds the object's data. When changes occur to an object, the VRP sends a message to the DRP to update the object's data. The VRP also sends the DRP a list of peers who are interested in the object, prompting the DRP to disseminate changes to these peers. When a VRP changes due to boundary changes, a message is sent to the DRP to update the VRP of the object [7]. As one can see, an object's DRP is not necessarily the owner of the region the object is in, thus eliminating the problem of frequent memory operations.

C. Latency

Latency (colloquially known as lag) is a key problem in games. It directly relates to the quality of experience of the player. If the game is not responsive (laggy), players experience visual artifacts and delayed feedback, which is disruptive and frustrating for the experience. This is especially true in competitive games, where the player's reaction time is crucial to winning the game. In this section, we will discuss the problem of latency in peer-to-peer architectures and how it is mitigated.

In server-client architectures, latency is mostly dependent on the quality of the connection between the server and the client and the server's computational power. Thus, to improve connections between players and servers, developers may buy servers located across different geographical regions. This increases the probability that the servers are closer to the players, thus reducing the time it takes for messages to travel between the server and the client [2].

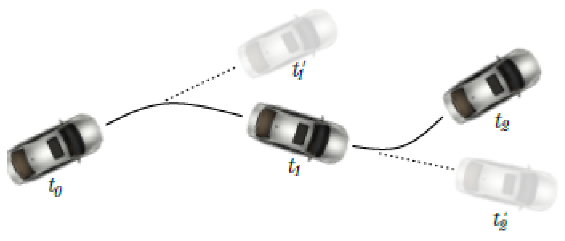

Fig. 3: This car starts at t_0, with predictions at $t_1^{'}$ and $t_2^{'}$ and real positions at t_1 and t_2 (from [17])

But in peer-to-peer architectures, the problem of latency is more complicated. The latency is dependent on the quality of the connection between the peers and the computational power of the peers. These parameters are out of the control of the developer, thus making the developer focus more on making sure their game can run well across a wide range of hardware and latencies (a similar problem to that in scalability).

Agarwal et al. proposes Htrae, a system that predicts the latencies between peers using geolocation data and matches them in the most (predicted) optimal fashion for matchmaking. Htrae also tunes the predicted latencies will real-time data in case the geolocation data was inaccurate. This scheme helps provide a guess of the most optimal network and update itself along the way. One should note that the system uses a central server to hold the data and route the network however [2].

We may also create super-peers to mitigate latency issues. A super-peer may not only arise because it has adequate computational resources, but also if connecting to the super-peer minimizes latency when compared to a direct connection [3]. However, we should still keep the redundant direct connections. If the super-peer fails, existing connections can help reduce the overhead of calculating a new super-peer. Additionally, comparing game states with multiple peers could determine whether the super-peer's state has been spoofed or not [3].

But despite this, a lot of times network latencies cannot be changed. A player may be stuck with poor internet service or a poor machine. It would be unfair to force players to upgrade their hardware or internet service too just have an enjoyable gameplay experience. Additionally, it may not be ideal to constantly share updates with peers, as this may consume bandwidth other apps may use.

Thus, developers have devised strategies to provide smooth gameplay by hiding latency issues. Termed as dead reckoning, the game predicts the current state of the game based on the most recently received state. In other words, an opponent's position on a player's client is estimated based on the most recent known velocity. When the opponent's true position is received, the client corrects the estimated position by blending the path the opponent takes (without blending, opponents snap to the true position, which looks jarring to the player) [17].

With dead reckoning, we reduce communication overhead, while having smooth gameplay. By enforcing a delay, we ensure only a few of messages are sent between peers in a period of time (time threshold). The extrapolation of positions becomes the primary method of updating the local game state, with the true positions being sent to correct these positions periodically. In fact, we may use a space threshold to neglect updates when predicted positions are close enough to the true positions [11].

Care must be taken when deciding the time and space thresholds for message latencies and update frequencies respectively. First, if thresholds are too long, the predicted positions may be too far from the true positions. When smoothing these positions on the local game state, the motion may abruptly accelerate to the desired position, breaking the smoothness goal. But if they are too short, we send unnecessary updates to peers, defeating the purpose of dead reckoning [11].

Regarding space thresholds, Jaya et al. combines the interest management concept mentioned above with dead reckoning. Like before, the space is partitioned into subregions and each peer is only concerned with updates from peers within their AOI and responsible for disseminating changes within their own region to surrounding peers. However, opponents that are closer or within the peer's own region are updated more frequently than those further away. This uses the intuition that players are more visually concerned with nearby opponents than those afar, meaning we can forego accuracy for moving objects that are far away [11]. This scheme allows for better bandwidth usage by reducing the messages we send to those that are critical.

In addition, in games with interactable objects, our local game state must also determine what to do when opponents supposedly collide with objects. Because our local state game state is an estimation, triggering a collision via the game engine may lead to inaccurate results if the opponent didn't collide with the object in the true game state. Thus, there must be a mechanism of ensuring how to handle dead reckoning when collisions are involved.

Walker et al. resolves these issues with their predictive dead reckoning system, which uses neural nets to predict the opponent positions and collisions. They describe the prediction as ”approximating the game engine”, which can be easily trained giving a numerous amount of training scenarios. They divide their algorithm into 3 cases, one for when opponents are close, one for when opponents are far, and one for when opponents are (or are anticipating) a collision. Each case is handled by a neural network that has varying inputs/outputs to tailor the accuracy for that case. For instance, the far case uses the most recent state, whereas the close case uses the three most recent cases in temporal order. To handle collisions, the system uses a neural network to predict the collision response rather than letting the physics engine handle it. This allows for smoother blending towards the actual aftermath with a specific obstacle. After all these are handled, the path is blended using linear interpolation which takes into account the car's

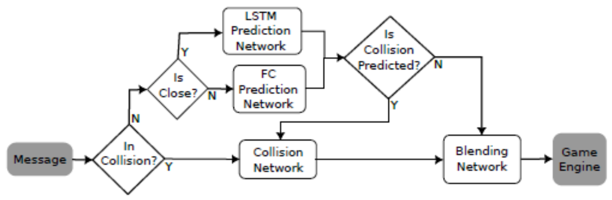

Fig. 4: The solution uses 3 neural networks, one for each case. A collision network is used for collision responses and an LTSM or FC network is used for position predictions depending on proximity (from [17])

dynamics [17].

III. Commercial Games and Remarks

After reading through many articles published by game journalists, I can conclude that the research I have just discussed and industry practices both verify and contradict each other. Specifically, it turns out that peer-to-peer architectures are widely used for peer-to-peer architectures for 2 player games and fighting games and server-client architectures used for games that involve many players at a time. This goes against the claims made in the scalability section, but verifies the points made in the latency section. But the reasoning behind these practices are sound.

First off, I found most games still opt for server-based matchmaking [14]. However, the connections between players could still be peer-to-peer. For instance, some popular games published by Nintendo such as *Splatoon* and *Mario Kart 8 Deluxe* use NEX, a library that provides a game server and API for matchmaking players into peer-to-peer sessions [15]. This is a good example of a hybrid architecture, where the server is used for matchmaking and the game is serviced with peer-to-peer. As seen before, with centralized matchmaking, we can utilize extra data such as geolocation data to predict the best connections between players [2].

Next, peer-to-peer architectures work best for 2 player games, especially fighting games. 2 player peer-to-peer connections are just direct connections, so the latency is minimized assuming a good connection. Additionally, synchronization is not a big issue because there are only 2 parties involved (unlike many in MMOGs). Finally, it is harder to cheat because both clients are running the same game state logic, and the only necessary data each is sending each other is input (one may argue that the inputs may be spoofed, but in fighting games, where precision and combos are key, I don't think that will help anyone) [13].

Finally, most commercial MMOGs are server-client and the existence/development of peer-to-peer MMOGs are limited to research. In practice, it is simpler to scale by buying machines, even though the price is steep. As seen above, with peer-to-peer MMOGs, although there is no need to buy machines, development costs increase because the problem and algorithms become more and more complex as the number of peers increase. Perhaps the biggest reason peer-to-peer MMOs are not mainstream is due to security. Especially in MMOGs with microtransactions, the virtual economy is at risk if the game state is not protected. In server-client architectures, security is guaranteed because the server holds the ground truth of the world. But in peer-to-peer, attackers are given access to the client, which also holds a part of the game state, making it easier for cheaters to gain unfair advantages [16]. To end, it is more profitable for the game developer (given enough money) to buy servers and maintain a secure environment for their game than to develop a secure peer-to-peer MMOG.

IV. Conclusion

In this paper, I have discussed the problems of connectivity, scalability, and latency in games and how peer-to-peer architectures may solve these problems. I have also discussed the feasibility of these solutions in commercial games and why some are not mainstream. I concluded that peer-to-peer architectures are a viable solution for games with 2 players, but are not practical for games with too many players. The main reason for this is due to the complexity of the algorithms and the security risks that come with peer-to-peer architectures. If given more time, I would've researched into synchronization and security algorithms, as well as other alternative architectures for peer-to-peer games. I hope that in the near future, we start seeing more studios implement peer-to-peer architectures in their games, so we can verify whether the assumptions made in research are true, and continue directing our focus towards enriching the online multiplayer gaming experience for all players.

References

[1] Video game market size, share and growth report, 2030.

[2] Sharad Agarwal and Jacob R Lorch. Matchmaking for online games and other latency-sensitive p2p systems. In *Proceedings of the ACM SIGCOMM 2009 conference on Data communication*, pages 315–326, 2009.

[3] Yousef Amar, Gareth Tyson, Gianni Antichi, and Lucio Marcenaro. Towards cheap scalable browser multiplayer. In *2019 IEEE Conference on Games (CoG)*, pages 1–4. IEEE, 2019.

[4] Krishan Arora. Council post: The gaming industry: A behemoth with unprecedented global reach, Nov 2023.

[5] Michał Boroń, Jerzy Brzeziński, and Anna Kobusińska. P2p matchmaking solution for online games. *Peer-to-peer networking and applications*, 13:137–150, 2020.

[6] Eliya Buyukkaya and Maha Abdallah. Data management in voronoi-based p2p gaming. In *2008 5th IEEE Consumer Communications and Networking Conference*, pages 1050–1053. IEEE, 2008.

[7] Eliya Buyukkaya, Maha Abdallah, and Romain Cavagna. Vorogame: a hybrid p2p architecture for massively multiplayer games. In *2009 6th IEEE Consumer Communications and Networking Conference*, pages 1–5. Ieee, 2009.

[8] Raluca Diaconu and Joaquín Keller. Kiwano: Scaling virtual worlds. In *2016 Winter Simulation Conference (WSC)*, pages 1836–1847. IEEE, 2016.

[9] Shun-Yun Hu, Shao-Chen Chang, and Jehn-Ruey Jiang. Voronoi state management for peer-to-peer massively multiplayer online games. In *2008 5th IEEE Consumer Communications and Networking Conference*, pages 1134–1138. IEEE, 2008.

[10] Shun-Yun Hu, Jui-Fa Chen, and Tsu-Han Chen. Von: a scalable peer-to-peer network for virtual environments. *IEEE Network*, 20(4):22–31, 2006.

[11] Iryanto Jaya, Elvis S Liu, and Youfu Chen. Combining interest management and dead reckoning: a hybrid approach for efficient data distribution in multiplayer online games. In *2016 IEEE/ACM 20th International Symposium on Distributed Simulation and Real Time Applications (DS-RT)*, pages 92–99. IEEE, 2016.

[12] Bjorn Knutsson, Honghui Lu, Wei Xu, and Bryan Hopkins. Peer-to-peer support for massively multiplayer games. In *IEEE INFOCOM 2004*, volume 1. IEEE, 2004.

[13] Mauve. "understanding fighting game networking", Oct 2021.

[14] Florian Metzger, Stefan Geißler, Alexej Grigorjew, Frank Loh, Christian Moldovan, Michael Seufert, and Tobias Hoßfeld. An introduction to online video game qos and qoe influencing factors. *IEEE Communications Surveys & Tutorials*, 24(3):1894–1925, 2022.

[15] Author OatmealDome. Splatoon 2's netcode and matchmaking: An in-depth look, Aug 2022.

[16] Gregor Schiele, Richard Suselbeck, Arno Wacker, Jorg Hahner, Christian Becker, and Torben Weis. Requirements of peer-to-peer-based massively multiplayer online gaming. In *Seventh IEEE International Symposium on Cluster Computing and the Grid (CCGrid'07)*, pages 773–782. IEEE, 2007.

[17] Tristan Walker, Barry Gilhuly, Armin Sadeghi, Matt Delbosc, and Stephen L Smith. Predictive dead reckoning for online peer-to-peer games. *IEEE Transactions on Games*, 2023.

[18] Feijie Wu, Ho Yin Yuen, Henry Chan, Victor CM Leung, and Wei Cai. Facilitating serverless match-based online games with novel blockchain technologies. *ACM Transactions on Internet Technology*, 23(1):1–26, 2023.

[19] Amir Yahyavi and Bettina Kemme. Peer-to-peer architectures for massively multiplayer online games: A survey. *ACM Computing Surveys (CSUR)*, 46(1):1–51, 2013.