Networking for Large-Scale Fast-Paced Multiplayer Games

Blake J. Bellamy
Division of Science and Mathematics
University of Minnesota, Morris
Morris, Minnesota, USA 56267
bella148@morris.umn.edu

ABSTRACT

Networking issues are plaguing large-scale fast-paced multiplayer games. There are many problems that contribute to making networking challenging for this genre of multiplayer game. Such problems include latency (lag), order of information arrival, server client architecture, and the algorithms handling all of the aforementioned situations. These algorithms include lag compensation, action synchronization, dead reckoning, workload balancing, and networking structure. Some algorithms like lag compensation solve general problems, while others such as dead reckoning solve specific ones. Having a solid framework to handle issues that arise from networking issue is important for any multiplayer game in this genre in order to display information to a client accurately and timely.

Keywords

Networking, Dead Reckoning, Peer-to-peer, Multiplayer

1. INTRODUCTION

Multiplayer online games rely upon infrastructure to communicate the state of the game between users. Compared to a single-player game, where game state can be instantly known, the details of game state in multiplayer games can vary for different users because of the time it takes for data to pass between computers. It is important to address these communication issues to make the games as fair and fun as possible [3].

With multiplayer networking over the internet, many factors influence the effectiveness of the multiplayer networking in the game. The networking strategies and implementations, or net-code, impacts the game experience for users. These strategies and implementation all deal with handling user data. While developers of video games are not able to control all elements of internet networking, such as connection speed and connection stability, there are efforts that are put in place to minimize the negative impacts of network-borne issues.

In this paper, we begin by providing some background information about networking and games. Then, we provide examples of various problems and potential solutions to the these problems. Each solution will include a brief overview of how the solution functions and how the solution contributes to bettering a fast-paced multiplayer experience. We conclude by examining both general-use and specific solutions, as well as offering suggestions about what to consider when making implementation choices.

UMM CSci Senior Seminar Conference, April 2020 Morris, MN.

2. BACKGROUND

2.1 Latency

Latency, or simply lag, is the time that any piece of data takes traveling from one destination to another. In the case of multiplayer video games, latency is the time it takes for a piece of data to leave a user's device (*client*), reach its destination, and then return to the user's device. For multiplayer games, a common destination is usually a *server*, a specialized computer tasked with simulating the game world and distributing data to any clients connected to it.

In a game played over a network, in the time it takes data to reach the server and return to the client, the state of the game may have significantly changed. This is especially true in fast-paced games. In the context of a shooter game, Player A could fire at Player B (whose client is connected to the same server). Player A's action is then sent to the server. Then, the server sends the message to Player B's client. Finally, Player B is shot. All of this communication between the clients and the server happens in the span of 400 milliseconds [5]. There must then also exist a method to compensate for aforementioned fluctuations in latency. Different inputs may arrive to the server or clients at different times, resulting in actions being taken out of order [8]. A notion of action order is important for relaying accurate inputs and game states. All of these factors contribute to the satisfaction of playing the game [3].

A client with high latency will have its data sent and received from a server more slowly than a client with low latency. Client data can contain anything from a user's position in the game world, to any action a user has taken, such as shooting. The physical distance between a client and a server also contributes to higher latency, as the data must move longer distances through the network cables laid in the ground to reach their destination [6, 7]. Latency values may also fluctuate during hours where many users are on the internet, referred to as network congestion. Latency is an inevitability with anything involving networking, and years of work has gone into attempts to compensate for it. In section 3, we describe several methods used to compensate for the negative impacts of latency in large-scale fast-paced multiplayer games.

2.2 Workload Balancing

For clients to communicate in a multiplayer game, each client must have a notion of sending its own relevant information to another client, or to a server, which will then distribute it to other connected clients. As the number of clients increases, so too does the amount of information being sent and distributed. If the incoming data from other clients is too much for a client or a server to process, that client or server may become overloaded, resulting in a performance drop, or a forced shutdown [2]. To prevent such an issue, one way to minimize the amount of data coming to a server or client is to trim down data to only the most relevant parts. Having a couple dozen clients sending everything they know about the current state of the game world would be taxing on the network. To avoid this type of overload, implementation of a method to trim the data being sent over the network down to its most important parts would be necessary [1]. In section 4, we describe two approaches for balancing workload to avoid overloading the network.

2.3 Network Structure

Traditionally, at a given time a fast-paced game has 32-64 players connected to a dedicated server [6]. The dedicated server locations also contribute to the lag situation, as most of them are hosted in large hot spots across the world. Being far away from these servers results in a higher latency, as your inputs have to travel the physical distance to reach them [7]. The structure of the network also effects how much data can be transmitted over the network at any given time, referred to as bandwidth. The more clients connected to a server, the more bandwidth required to send data to all those clients. In section 5, we describe ways clients are connected to each other and how information is sent between them, including challenges that can arise in certain configurations and ways to address those challenges.

3. ADDRESSING LAG

3.1 Traditional Lag Compensation

Lag compensation is a method implemented in most modern day fast-paced shooting games to, as its name implies, offers a method of compensating for the amount of time data takes to reach a server, and then the other players. A good example of the importance of lag compensation can be observed in multiplayer shooting games.

Consider two players on separate computers, call them Player A and Player B. Let Player A have low latency, and Player B have high latency. If Player A is walking in a straight line, and Player B takes a shot at Player A, by the time the information that Player B shot at Player A reaches the server, Player A has now reported they are in a different position, and therefore the shot hits nothing. Compensating for this latency is accomplished by storing players previous locations and using the amount of latency as player has to roll back time to when an action was taken, see Figure 1. This is referred to as traditional lag compensation (TLC). This allows the server to then calculate if an action was successful or not.

In the context of a shooting game, this approach favors the shooter, since whether the shot is successful is entirely dependant on what the shooter saw when the shot was fired. This results in a situation called "shot behind corner", where a less lagged Player A is shot at from behind a wall by a much more lagged Player B who is currently seeing Player A's rewound hit-box. [5]

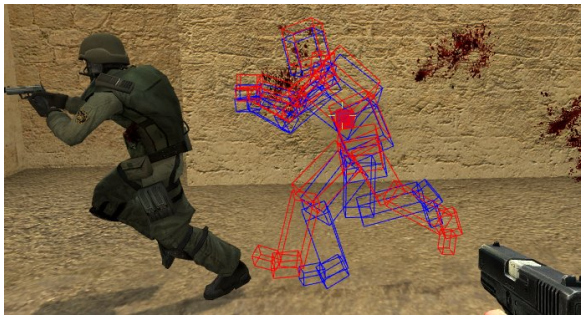

Figure 1: A player's rewound hit-box based on latency

3.1.1 Advanced Lag Compensation

To go one step above TLC is to solve the problem of being shot behind corners. Using an advanced lag compensation (ALC) presented by Lee and Chang [5], allows clients to dispute actions based on if they detect a shot behind cover situation. In the event such a situation occurs, the ALC will send a dispute message to the server. From there, the server will further evaluate if the dispute is trustworthy, resulting in either a hit confirmed, a hit denied correctly, or a hit denied incorrectly that is irrefutable. [5] By allowing the client to refute actions, incidents such as being shot behind cover dropped by 94.1% [5]. There is also merit for the server being able to reject a client dispute. Some players cheat in video games, and modify their own game to send false information to the server. Having an authoritative algorithm to check the validity of a dispute is desirable for fairness of the game.

3.2 Synchronization of Actions

While lag compensation helps with bridging the gap between latency, it does not help with the synchronization of actions. Because data take time to travel from client, to server, and to other clients, the actions sent can become desynchronized from the time they actually occur, and in some cases, even occur out of order. For fast-paced multiplayer shooting games, the most relevant issues involving synchronization of actions comes in the form of action start and completion time. Simply put, when a player enters a command to move forward, by the time that data reaches the other players in the game, some time has passed (Figure 2). This gap in time is what causes the desynchronization between the player who moved and all the other players, a gap that is proportional to the amount of latency a player has. Much like in 3.1.1, occurrences involving being shot behind walls would be common if there were not a solution. In addition to being shot behind cover, synchronization of actions would also apply to actions that should not exist anymore. As an example, if a player is killed and the server validates that they are indeed slain, then any further incoming actions from that player must be voided, as their actions would not be able to be taken since they are dead.

3.2.1 Local Lag

One approach to solving this issue is to introduce local lag to every player in the game. Local Lag is a novel solution

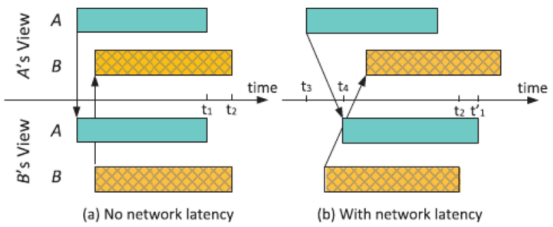

Figure 2: Action synchronization with and without latency [8]

that introduces a slight delay to every action a player takes client side. This allows their data to reach all the other players at the same time their action has taken to reach other players, thus allowing for a consistent synchronization [8]. Local lag is not without its own drawbacks. By delaying the inputs of players locally, the game can begin to feel sluggish, especially as the latency grows higher and higher [8]. Xu and Wah describe this sluggishness as a just-noticeable-difference (JND) threshold, were when introducing methods such as local lag, there exists a threshold where action delays can be perceived by the player[8].

3.2.2 *Perception Filtering*

Where as local lag adjusts the timing of when actions occur, perception filtering modifies the duration of actions happening. When a player takes an action, the perception filter either extends or shortens that action by an amount proportional to latency to preserve action synchronization. Much like local lag, this solution is not without its own JND threshold problems. In a fast-paced multiplayer shooting game, small taps of movement to readjust a players position could be shortened or extended farther than intended [8], which could result in a player sticking out from behind cover or not being covered enough on another player's screen.

3.2.3 *Combing Local Lag and Perception Filtering*

While on their own, each solution detailed in 3.2.1 and 3.2.2 can lead to too much variance in the JND threshold, however, by combining the two techniques a more favorable outcome surfaces. Xu and Wah were able to combine the two techniques into one combined technique which allowed for a strong synchronization with very few JNDs [8]. This approach solves both problems with local lag and perception filtering by covering up each other's weaknesses. Local lag allows for both actions to complete at the same time, and in the same positions which perception filtering erred on. Perception filtering allows for actions to start with less local lag by extending the actions to remote players, allowing for less sluggish responses, compensating for a shortcoming of local lag (See Figure 3). Compared to its individual components, the combined solution had a considerable improvement with the fewest JNDs (See Figure 4) [8]

4. WORKLOAD BALANCING

Just like web-servers, multiplayer video games must connect either to a server, or in a peer-to-peer network to communicate and play together. Also like web-servers, video game players and servers are vulnerable to being overloaded with information if not handled correctly. The most common

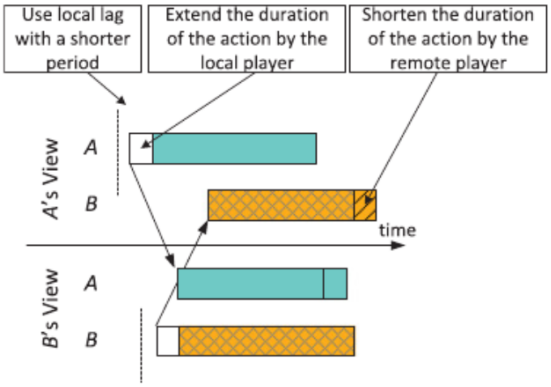

Figure 3: Combination of local lag and perception filtering. [8]

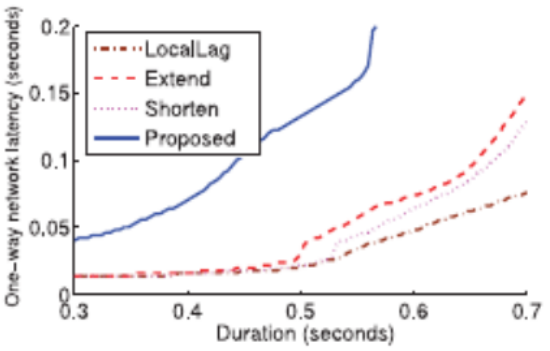

Figure 4: Comparison of the individual components and the combined results. [8]

occurrence of this happening is when a server becomes full of too many players. Each player will be sending information relevant to itself to the server, which is then responsible for handling and distributing all of that information to all the other connected clients. In the event that this happens, the game is likely to slow down, disconnect excess players unexpectedly, or even crash if the stress is not alleviated. It is important that the workload is distributed or information be filtered to avoid an overload such as this.

4.1 Zone Splitting

In most modern day multiplayer games, the developers of the game host dedicated servers for their players to play on. They are designed as such so that many different matches or zones can be hosted on the same server, depending on the context of the game being hosted. If one server is experiencing too many matches or players, the game will slow down as detailed above, or in the worst case, crash. To avoid this, a server can implement a method known as zone adding/shedding (ZAS) [2]. A ZAS is an attempt by the server to migrate players from one overloaded server to another that can accommodate them, allowing for the overloaded server to function normally again. This is most useful for areas in which a large amount of players will be participating in, while not needing to interact with one another. ZAS is not without an inherent drawback, and by splitting

all these zones, as players disconnect, there ends up zones with a small amount of players, making interactions less frequent. To solve this problem, there exists another method called Split Zone Rejoining (SZR), in which the players of two zones are merged into one, freeing up one of the servers for more ZAS operations [2]. A SZR will never join two zones together in a way that will increase workload, allowing for more opportunities for ZAS to occur.

4.2 Dead Reckoning

In addition to easing server load, the amount of information being sent is also important to balance the workload. Consider the information about every player in a game: a player has an x, y, and z coordinate in a 3D space, a velocity, a model, and an animation. Sending all that information to the server may be fine with a few players, but as the player count scales up, so too does both bandwidth consumption and computation needed[1]. Dead Reckoning is a solution implemented to filter out unneeded information by only sending important state information from a player, and having each client calculate the state using a set of predefined algorithms. For example, if Player 1 was standing at point A with a velocity of 10 miles per hour, other players could calculate Player 1's new position, point B, by using their velocity, cutting down on the information Player 1 would have to send to the server. By filtering down the amount of information to send, and having clients calculate the positions of players, the server has more resources available for computing. Dead reckoning is not without drawbacks though. Dead reckoning works best when objects are moving in predictable paths, and human players tend to move in unpredictable manners [1]. Sharp and or sudden changes in an object's path cause dead reckoning algorithms to incorrectly predict paths. Dead reckoning calculations are also required for a server to preform as well for the purpose of checking the validity of a calculated path; a common instance of this being players reporting themselves moving through walls.

5. NETWORK STRUCTURE

Network structure is just as important as any algorithm used to handle it. Network structuring refers to the medium in which players are connecting to one another. Traditionally, players in a multiplayer game will connect to a server, wherein the server acts as any other server would act and relay data and inputs between all the connected players. While not a bad solution, there are cases in which a traditional server structure like this struggles, the main component of this being physical distance. Traditionally in the United States, servers like these are hosted in high density population areas, or areas that span large amounts of the country, like in Dallas, Texas. The issue here lies in that players who are not near these hot-spots will have higher latency than those closer due to the nature of networking. Players who may be within a few miles of one another will have to connect to a hot-spot server location hundreds of miles away just to play with each other, resulting in needless higher latency.

5.1 Peer to Peer Network

A peer-to-peer (P2P) network is one in which players connect and send information to one another directly, rather than connect to a central server. This makes every user responsible for information to other connected users. Each user in a peer-to-peer network is referred to as a node, and these connections of nodes make up the network. In the scope of a large-scale fast-paced, peer-to-peer networks function best when the connected users are close to one another physically. However, traditional peer-to-peer networks have a strong drawback for our scope; they do not scale well as the number of users goes up. Every user connecting directly to every other user can quickly overload user computers if too many connections are present [6]. This is a glaring problem that must be overcome if it is to fit in the scope of our genre.

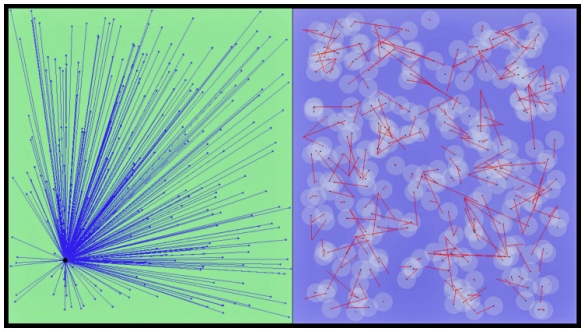

Figure 5: On the left is a traditional server model. On the right is a peer-to-peer hybrid clustering model. [7]

5.1.1 Hybrid Clustering

Hybrid clustering is a networking model that uses both peer-to-peer and a more traditional server, called an edge server, to communicate, allowing for the benefits of both networking structures. Hybrid clustering uses network topology rather than the physical location of the clients and servers [7]. Rather than all clients in a game session connecting to one server, or directly to each other, hybrid clustering splits up clients into 'pools' based on participation in common activities in game. These pools are peer-to-peer connections between the players in the pool 5. Players may freely enter and exit these pools of other players based on their activities in game. The pools then connect to a traditional server through one of the players in the pool, which then communicates the state of the pool to the server. This significantly reduces the latency involved when players are involved in common gameplay, and the connection between a client (in this case a pool) and the server, however, it does not preform as well when players are not able to cluster efficiently (See Table 1).

Table 1: Comparing Latency Without Hybrid Clustering and With Hybrid Clustering Across Three Types of Connections [7]

Connection type	Latency (Without Hybrid Clustering)	Latency (With Hybrid Clustering)
Common Gameplay	15.2 ±5.1 ms	8.4 ±4.0 ms
Server to Client	7.6 ±3.7 ms	6.3 ±3.5 ms
Area-of-Interest	15.2 ±5.1 ms	18.5 ±7.6 ms

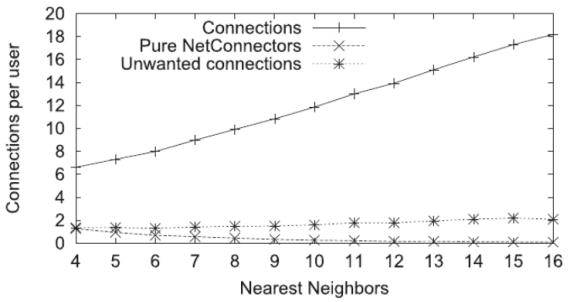

Figure 6: A graph showing the number of unwanted connections stays low by connecting logically to nearby neighbors. [6]

5.1.2 Net Connectors

Net connectors are an improved solution to hybrid clustering. Much like hybrid clustering, net connectors work by connecting players that are nearby one another, or participating in the same activities. However, where they differ is how connections between 'neighbors', players nearby to one another, are established. Net connectors use a publish/subscribe model for user connection. This model allows for users to subscribe to other users, and then publish information only to those subscribed users. This model allows net connector to follow the principle that each user shoulder manage their own responsibilities and connections [6]. However this independent user connection approach is not without problems. An issue with this is that discovering neighboring players in your network becomes difficult, as there is no coordinating central instance or server to put players into pools like in hybrid clustering. To solve this, users periodically send out requests to all known neighbors with lists of all their known neighbors. This creates a 'neighborhood' of connections, and forms the basis for the connection scheme. From there, each individual user will logically determine if a connection should be made based on similar criteria to hybrid cluster, thereby limiting the amount of connected players at once. Results from simulations showed that using this networking scheme provided a peer-to-peer network that was scalable (See Figure 6). Using this system allows for the discovery of new neighbors while at the same time being keeping network load low and having a user maintain its own connection information.

5.2 Cheat Prevention in P2P Networks

While peer-to-peer networking seems promising for our scope, there exists a large security flaw. Without a central server to process and fact check all of the incoming data, peer-to-peer networks open up the opportunity for cheaters to modify game data. This is due to the fact that in a peer-to-peer network, users are handling game logic and data directly as it is passed around. This fact is largely why peer-to-peer networks are less common than traditional client-server networks [4]. There are two common inherent problems in peer-to-peer networking that must be handled to maintain a fair game state for all connected users. The first is a user exploiting misplaced trust, which involves a user manipulating software or data locally on their machine. The second is a user exploiting a lack of secrecy, in which a user has access to all data regardless of if a user should have access to it.

5.2.1 Exploiting Misplaced Trust

In our scope, exploiting misplaced trust would mean to modify game files or game data in order to give a user an advantage. A simple exploit using this misplaced trust would be to modify a character's own health value so it could take more damage before dying, or even to give it infinite health. These types of exploits are made possible due to no central server fact checking information sent to other players. In order to counteract this, a system must be put in place for all users in a peer-to-peer network to agree on what is valid and what is not. To handle this, instead of blindly trusting a single user, all users will vote on what the correct game state should be. This process is what Kabus and Buchmann refer to as Mutual Checking [4]. All users connected to a peer-to-peer network will broadcast their data to another node in the network referred to as a region controller. This region controller will then compare the different states, and choose the state with the least discrepancies from user to user as the accepted game state, which is then broadcast back to all the players in the network. This allows for connected users to agree on what the state of the game should be rather than blindly trusting information.

5.2.2 Exploiting Lack of Secrecy

The next exploit to address is exploiting the lack of secrecy within peer-to-peer networks. This type of exploit deals with using data normally not available to a user. An example of this exploit would be learning the positions of players without having the means to know. A simple example of this exploit is referred to as a 'wall hack', in which players are able to see other players behind walls. The wall hack exploit is made possible due to the fact that players are still reporting themselves in a certain position when sending their data to other clients, regardless of if other players are able to see them or not. Malicious users could then take that information, and modify their game to always display other players regardless. To solve this, Kabus and Buchmann propose that sensitive information should be encrypted for all except the intended subscribers, as well as limited the saving of data locally. That way, all the users connected can still preform Mutual Checking, while sensitive information is kept sensitive.

6. CONCLUSION

While each of these methods help to compensate for the effects of multiplayer gaming over a network, some methods are more general use, while others handle specific situations. For example, dead reckoning is an algorithm that can be used across all types of games in the genre of large-scale fast-paced multiplayer games, whereas advanced lag compensation handles a specific situation in shooter games. It is therefore important to consider the needs of the game when implementing methods for compensating gaming over a network.

The use of a peer-to-peer network is not as commonly used as its server network counterpart in this genre [4]. Many of the algorithms discussed in this paper can only be applied to peer-to-peer networking. Although concerns about lack of fairness is a common opposition to using peer-to-peer networks [3, 4, 6], the latency and workload benefits of a peer-to-peer network in this genre far exceeds any additional computational load incurred by applying some of the cheat prevention techniques presented in this paper. For this

reason, consider using a peer-to-peer network for games in this genre.

Acknowledgments

I would like to thank professors Kristin Lamberty, Peter Dolan, and Hussam Ghunaim for their invaluable feedback and guidance on this paper. I would also like to thank Andy Korth for his feedback.

7. REFERENCES

[1] Y. Chen and E. S. Liu. Comparing dead reckoning algorithms for distributed car simulations. In *Proceedings of the 2018 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, SIGSIM-PADS '18, page 105–111, New York, NY, USA, 2018. Association for Computing Machinery.

[2] S. Farlow and J. L. Trahan. Periodic load balancing heuristics in massively multiplayer online games. In *Proceedings of the 13th International Conference on the Foundations of Digital Games*, FDG '18, New York, NY, USA, 2018. Association for Computing Machinery.

[3] E. Howard, C. Cooper, M. P. Wittie, S. Swinford, and Q. Yang. Cascading impact of lag on quality of experience in cooperative multiplayer games. In *Proceedings of the 13th Annual Workshop on Network and Systems Support for Games*, NetGames '14. IEEE Press, 2014.

[4] P. Kabus and A. P. Buchmann. Design of a cheat-resistant p2p online gaming system. In *Proceedings of the 2nd International Conference on Digital Interactive Media in Entertainment and Arts*, DIMEA '07, page 113–120, New York, NY, USA, 2007. Association for Computing Machinery.

[5] S. W. K. Lee and R. K. C. Chang. Enhancing the experience of multiplayer shooter games via advanced lag compensation. In *Proceedings of the 9th ACM Multimedia Systems Conference*, MMSys '18, page 284–293, New York, NY, USA, 2018. Association for Computing Machinery.

[6] P. Mildner, T. Triebel, S. Kopf, and W. Effelsberg. Scaling online games with netconnectors: A peer-to-peer overlay for fast-paced massively multiplayer online games. *Comput. Entertain.*, 15(3), Apr. 2017.

[7] J. N. Plumb, S. K. Kasera, and R. Stutsman. Hybrid network clusters using common gameplay for massively multiplayer online games. In *Proceedings of the 13th International Conference on the Foundations of Digital Games*, FDG '18, New York, NY, USA, 2018. Association for Computing Machinery.

[8] J. Xu and B. W. Wah. Consistent synchronization of action order with least noticeable delays in fast-paced multiplayer online games. *ACM Trans. Multimedia Comput. Commun. Appl.*, 13(1), Dec. 2016.