Peer-to-Peer Architectures for Massively Multiplayer Online Games: A Survey

Article *in* ACM Computing Surveys · October 2013

DOI: 10.1145/2522968.2522977

Amir Yahyavi
McGill University

Peer-to-Peer Architectures for Massively Multiplayer Online Games: A Survey

AMIR YAHYAVI and BETTINA KEMME, McGill University

Scalability, fast response time, and low cost are of utmost importance in designing a successful massively multiplayer online game. The underlying architecture plays an important role in meeting these conditions. Peer-to-peer architectures, due to their distributed and collaborative nature, have low infrastructure costs and can achieve high scalability. They can also achieve fast response times by creating direct connections between players. However, these architectures face many challenges. Distributing a game among peers makes maintaining control over the game more complex. Peer-to-peer architectures also tend to be vulnerable to churn and cheating. Moreover, different genres of games have different requirements that should be met by the underlying architecture, rendering the task of designing a general-purpose architecture harder. Many peer-to-peer gaming solutions have been proposed that utilize a range of techniques while using somewhat different and confusing terminologies. This article presents a comprehensive overview of current peer-to-peer solutions for massively multiplayer games using a uniform terminology.

Categories and Subject Descriptors: C.2.4 [**Computer-Communication Networks**]: Distributed Systems; K.8.0 [**Personal Computing**]: Games

General Terms: Algorithms, Design, Human Factors

Additional Key Words and Phrases: Peer-to-peer networking, massively multiplayer online games, interest management, network overlays, multicasting, replication, fault tolerance, consistency control, incentives, cheating, commercial applications

ACM Reference Format:
Yahyavi, A. and Kemme, B. 2013. Peer-to-peer architectures for massively multiplayer online games: A survey. ACM Comput. Surv. 46, 1, Article 9 (October 2013), 51 pages.
DOI: http://dx.doi.org/10.1145/2522968.2522977

1. INTRODUCTION

Massively Multiplayer Online Games (MMOGs) are among popular online technologies that produce billions of dollars in revenues and introduce several new and interesting challenges. One of the main attractions of these games lies in the number of players that participate in the game. The more players that are in the game world, the more interactive, complex, and attractive the game environment will become. Successful games such as *World of Warcraft* with nearly 12 million subscriptions [Blizzard 2011] have to provide a truly scalable game world while maintaining responsiveness.

To better understand MMOGs, we first need to give a definition. Video game is usually defined as an electronic game that is played by a controller and provides user interactions by generating visual feedback. A multiplayer game is a game played by several players. Players can be simply independent opponents or they can play in

A. Yahyavi was funded by NSERC Strategic Grant STPGP/350626-2007.

Authors' addresses: A. Yahyavi (corresponding author) and B. Kemme, School of Computer Science, McGill University, Montreal, QC H3A 0G4, Canada; email: amir.yahyavi@cs.mcgill.ca.

DOI: http://dx.doi.org/10.1145/2522968.2522977

teams. They can play against each other or can play against the game, that is, opponents that are controlled using Artificial Intelligence (AI). An MMOG is a game capable of supporting hundreds or thousands of players and is mostly played using the Internet. Many games such as *World of Warcraft* [Blizzard 2011], *EVE Online* [EVE 2011], and *Final Fantasy XI* [FFXI 2011] have shown that MMOGs are a thriving business industry. For example, *Star Wars: The Old Republic* was able to achieve one million subscribers in three days after launch.[1] *Second Life* [SecondLife 2011], launched in 2003 by Linden Lab, is the most famous social virtual world with more than 16 million registered users. The emergence of social games (such as *Farmville* and *Mafia Wars*[2]) with millions of subscribers [Zynga 2011] as well as mobile games that are played on smartphones, and the popularity of handheld devices such as Sony PSP [2011] and Nintendo DS [Nintendo 2011], lay the foundation for potential integration of social and mobile environments into massively multiplayer games [Iosup et al. 2010; Varvello and Voelker 2010].

MMOGs can produce huge network traffic and processing loads [Suznjevic and Matijasevic 2012; Chen et al. 2005b]. Thus, the main challenges in MMOGs are *scalability*, that is, providing support for thousands of players simultaneously, *consistency*, *security*, and *fast response time*, and usually all at the same time, otherwise customer satisfaction would be reduced. In the next sections we discuss these challenges and different solutions that have been proposed.

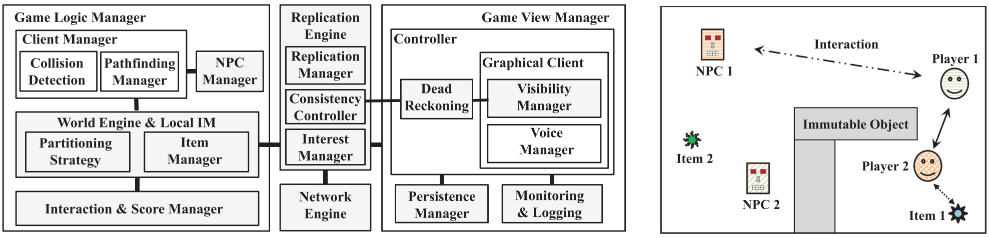
Client-server systems, where game execution and game state dissemination are completely controlled by the server, are currently the prevalent game architecture. However, peer-to-peer architectures can be beneficial for gaming infrastructures in several ways. If client nodes communicate directly with each other or perform part of the game state computation, server requirements in terms of computational power and network bandwidth can be significantly reduced. Even if the game execution remains completely controlled by servers, peer-to-peer technology can be used to coordinate multiple servers, such as maintaining distributed game state execution and management of server farms [Chen et al. 2005a] or federated servers [Iimura et al. 2004; Ahmed et al. 2009]. Cloud-based game streaming services based on content distribution networks [OnLive 2011] can also benefit from these architectures [Chien et al. 2010]. They can provide 3D streaming services where, similar to audio or video media streaming, 3D content is fragmented into pieces at a server, before it is transmitted, reconstructed, and displayed at the clients [Wu et al. 2009].

Peer-to-peer architectures have received a great deal of research attention in the recent past as they distribute computational and network load among peers, can potentially achieve high scalability, low cost, and good performance as will be discussed further. While most of our discussions are focused on multiplayer games, in particular MMOGs, many of these architectures can also be applied to other distributed systems such as distributed simulation environments [Fujimoto 2000], virtual worlds such as *Second Life* [SecondLife 2011], and other networked virtual environments [Blau et al. 1992; Kawahara et al. 2004; Keller and Simon 2003].

The remainder of this article is structured as follows. In Section 2 we study common game design principles used in most multiplayer games, and in particular MMOGs. In Section 3 we study and compare different architectures proposed for MMOGs. Next, we present in detail issues related to structure (Section 4), update dissemination (Section 5), interest management (Section 6), replication and consistency (Section 7), fault tolerance, availability, and persistence (Section 8) in peer-to-peer-based MMOG solutions. Section 9 discusses cheating. We explain how games, in particular P2P

[1] http://www.swtor.com/news/press-release/20111223.

[2] Zynga: 232 Million Monthly Players http://secfilings.com/searchresultswide.aspx?link=1&filingid=8022980.



(a) Multiplayer game components

(b) Object types and interactions

Fig. 1. (a) Different components of a multiplayer game, adapted from Mammoth [Kienzle et al. 2009] (adapted, ©ACM 2009), a massively multiplayer game research framework, are presented. The components discussed in this article are highlighted; (b) different game objects and their interactions. Players can interact with each other, objects, and NPCs.

architectures, are affected by cheating and discuss some of the security measures proposed for P2P-based games. In Section 10 we study different incentives for using P2P architectures by consumers and the industry. Furthermore, we discuss various applications and adoption models of P2P-based gaming architectures. Section 11 concludes the article.

2. DESIGN PRINCIPLES

Before getting into peer-to-peer architectures we first explain general concepts involved in designing a multiplayer game whether using client-server architectures or P2P. An overview of different components of a sample multiplayer game framework is shown in Figure 1(a). Here, we discuss the general concepts and execution patterns used in most multiplayer online games.

2.1. Object Types

In modern video games, the game world is usually made up of four types of components [Knutsson et al. 2004]: (1) *immutable objects*, such as landscape or terrain information, are usually designed and created offline and never change during the game. These objects are typically installed at the client and are initialized at the start of the game. (2) *characters* or *avatars* in the game world are controlled by the player using an input device. The avatar has a position in the game world and is usually allowed three types of actions: player updates, player-object interaction, and player-player interaction. (3) *mutable objects* such as food, weapons, and tools can be modified. For instance, players can interact with and/or use them in their interaction with other players. (4) *NonPlayer Characters* (NPCs), also called *bots*, are characters or avatars that are not controlled by a player but are usually controlled using AI. In the rest of this article, unless otherwise stated, *game objects* refer to avatars, mutable objects, and NPCs in the game. Object types are shown in Figure 1(b).

2.2. Player Interactions

Player interactions are typically divided into three categories: *player updates*, *player-object interactions*, and *player-player interactions* [Knutsson et al. 2004; Bharambe et al. 2006]. Player updates are interactions with the game world that only affect the player himself. Position updates and graphical updates to the player's avatar are examples of player updates. In a simple and unoptimized implementation, a large proportion of all player interactions can be position updates [Knutsson et al. 2004]. Player-object interactions are the interactions between a player and mutable objects in the game world. For example, picking up a health pack (adding it to the inventory) or consuming it are examples of player-object interaction. Player-player interactions are interactions between a player and other players in the game world. For example, attacking another player could decrease the other player's health and increase the experience points for the attacker. Player interactions with NPCs, based on the game design, can be considered either player-object or player-player interactions.

The type of interaction is important when dealing with consistency issues that arise from concurrent and conflicting updates to the same object. Also, most security and cheat prevention techniques only apply to certain types of interactions.

2.3. Object Replication

When a player joins a game, she receives an instance of the game world (it can be a limited view of the game world) that is made up of various types of game objects. Most game engines follow a primary copy replication approach. For each object and character there exists an authoritative copy, called *primary* or *master copy*. All other copies are *secondary copies* (also called *replicas*). Each player has, stored on his computer, copies of game objects which are of interest to the player. Any update to the object has to be first performed on the primary copy. How primary and secondary copies are distributed (e.g., primary copies might always reside on the server or might also be held by clients) depends on the game architecture. If a player wants to perform an update on an object for which she does not have the primary copy, she has to send the update to the primary copy. The holder of the primary copy decides whether to accept the update or not, and then sends the updated object to everyone that has a secondary copy, where the changes are applied.

The update dissemination mechanism is quite similar to publish-subscribe systems that have been widely studied [Castro et al. 2002b]. Every replica becomes a subscriber to the primary copy of the object and receives publications (updates) from the primary copy (the publisher).

2.4. Game Types and Latency Tolerance

We refer to the latency as the delay between execution of an update at the primary copy of an object and the replica receiving the object update. This latency is dependent on the architectural design and networking delays as will be discussed.

Various types of multiplayer and massively multiplayer games exist. In Real-Time Strategy (RTS) and Role Playing Games (RPGs) [Sheldon et al. 2003] the focus is more on game strategy rather than responsiveness. The player tells the avatar to do a possibly complex and long-lasting action, for example, go to a destination, and the avatar performs the requested action. In First Person Shooter (FPS) games, however, the player does short-lived and less complex actions, that is, the player actually does what he wants the character to do (for example, guides the avatar towards the destination) and as a result, higher responsiveness is required (see Armitage [2003] for *Quake III* latency requirements). Higher latencies than the tolerance threshold of the game have an adverse effect on playability of the game and user satisfaction. Based on the game type and design, games typically can tolerate latencies between 100 to 300 milliseconds (ranging from FPS games to RPGs) [Armitage 2003; Bharambe et al. 2006; Pantel and Wolf 2002b], however, games with higher latency tolerance exist (see Suznjevic and Matijasevic [2012]). Latency tolerance requirements have a dramatic effect on the architecture design for games. An architecture would only be feasible if it meets game latency requirements.

2.5. Bucket Synchronization and Frame Rate

Bucket synchronization [Lety et al. 1998] (also called *local lag* [Mauve et al. 2004]) is a method used to deal with latency and is used in most multiplayer games. Since network latency for each client is different and it is common for a primary copy to receive various update requests concurrently from different clients, most games deliberately *lag* behind in executing the events. This allows fairness despite latency variations and more control over update dissemination costs. This is in essence similar to Nagle's algorithm in IP networking. Games implement a discrete event loop (may also be referred to as *frame*) in which all actions (events) that have been submitted since the last execution of the loop are received, buffered, and then executed. The updates are then sent at the end of the loop. In addition, most game objects, including NPCs, have a *think* function which is executed in every game loop and determines the actions of the object in this loop. The game loop is usually executed 10 to 20 times every second; this frequency is sometimes referred to as *frame rate* [Bharambe et al. 2008] (not to be confused with graphical frame rate). Low frame rates can degrade the game play experience or even render the game unplayable. Note that based on the game design the number of frames shown to the player, that is, the graphical frame rate, may be equal to or different from this frame rate.

The lag should be chosen so that it allows enough time for the updates of different clients to be received. This helps ensure fairness for clients that might have worse connectivity, and controls the number of updates that have to be sent per second. At the same time, the lag has to be small enough not to be perceived by the players. In essence, a trade-off has to be found.

In this article we focus on the impact of architectures on networking, processing delays, and the resulting frame rate. We do not consider other delays such as those caused by graphical processing.

2.6. Bandwidth Requirements

The bandwidth requirement of MMOGs can be calculated based on average message size, update rate, and number of recipients (active players). Games with millions of (active) subscribers (e.g., WoW) have high bandwidth requirements due to their dynamic environment (e.g., *Second Life*), or high update rates (e.g., Quake) [Chen et al. 2005b; Suznjevic and Matijasevic 2012]. In addition, in order to have an acceptable game play, games should also accommodate occasional bursts in the game traffic that can be many times the average requirements. Such bursts happen due to sudden environment changes or battles. Moreover, even inside the same game different gaming activities, such as raiding versus trading in WoW, generate different traffic [Suznjevic et al. 2011] that can lead to different network requirements. Game servers typically deal with this by overprovisioning.

2.7. Interest Management

Interest Management (IM) [Benford and Fahlén 1993; Morse 1996] is an important mechanism used in many games, typically for scalability reasons. The idea is that players of a multiplayer game have only limited movement and vision capabilities. That is, players can only move a small distance in the game world in a given time interval. The players also have limited sensing capabilities, meaning that players can only interact with objects and other players in their vicinity [Makbily et al. 1999]. As a result, data access in games shows spatial and temporal locality. Utilizing this fact, interest management limits the amount of game state any player can access. That is, the player only receives the game state relevant to it, based on his position and vision in the game world. Interest management is important for scalability, as clients only need replicas of objects that are interesting for them, therefore keeping update and network overhead low. However, it can also be important as part of game semantics or to address other challenges such as cheating. IM plays an important role in how replication and communication between players are managed as will be discussed in the next sections, but first, we explain how it is implemented.

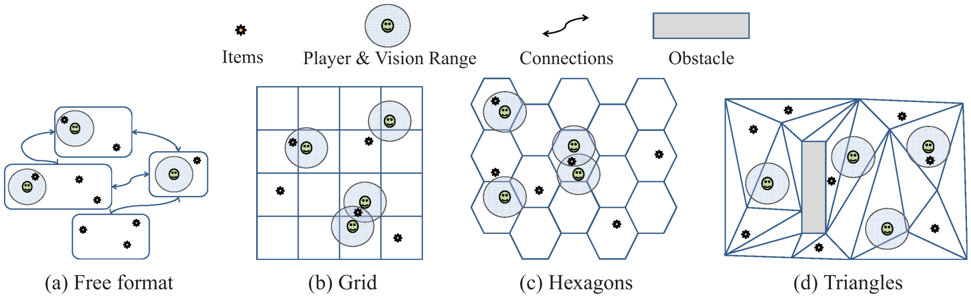


Fig. 2. Different game zoning mechanisms [Chen et al. 2005a] (adapted, ©ACM 2005); [Boulanger et al. 2006] (adapted, ©ACM 2006).

Space-based interest management is based on proximity and follows an *aura-nimbus* model [Benford and Fahlén 1993]. *Aura* is the bounding area around the player's avatar, while *nimbus* defines the area around the player that is visible to the player, that is, the player can perceive game objects located in this area. Nimbus is also often referred to as Area-of-Interest (AOI) [Knutsson et al. 2004; Schmieg et al. 2008], Domain of Interest [Morse 1996], Aura [Greenhalgh and Benford 2002], or awareness area [Keller and Simon 2003]. We mainly use the term AOI in this article. A player can typically only interact with objects and players in her AOI, and therefore, only needs to have copies of these objects. As a result, the necessary computational and network requirements are substantially reduced compared to maintaining copies of all objects.

Zoning. The most common mechanism for interest management is *zoning* where the game world is partitioned into smaller sections, called zones or regions, as depicted in Figure 2. Zoning approaches differ in the shape of their zones and how the AOI is mapped onto zones. In the simplest approach, the entire AOI resides in the same zone in which the player is located. In some systems, the player can simply interact with all objects in his zone, that is, his AOI is the entire zone [Berglund and Cheriton 1985; Lety et al. 1998]. A player's AOI changes only if he moves from one zone to another. In other approaches, the AOI is a subarea of the entire zone. Often, it is a fixed-radius circle (or sphere) around the player. When the player moves, his AOI moves accordingly. Therefore interest management has to determine for each game object in the zone whether it falls in the current AOI of the player. Figure 2(a) shows an example of this subarea approach.

More advanced interest management schemes allow the AOI to cover more than one zone. This is important for continuous worlds. Players at the borders of a zone should be able to see and interact with objects that are just across the zone boundary in a neighboring zone. Figures 2(b), (c), and (d) show examples where the AOI can cover several zones.

In regard to zone shapes, the game world can simply be split statically into grid cells as shown in Figure 2(b). Since this is a straightforward approach, many solutions use it [Chen et al. 2005a; Iimura et al. 2004; Cecin et al. 2004]. Hexagonal zoning (Figure 2(c)) has also been widely used for game architectures [Yu and Vuong 2005; Macedonia et al. 1995; Jaramillo et al. 2003] as well as other systems such as cellular networks. Hexagons have uniform orientation and uniform adjacency, meaning that players will always move to an adjacent zone. In addition, since most AOI mechanisms consider a circle, hexagons provide a good approximation.

Using triangulation, as shown in Figure 2(d), allows for taking obstacles in the game world into account. This can help in reducing the AOI, and thus the number of objects that have to be considered for interest management, leading to less object replicas to be maintained by the players, and less update messages that need to be sent [Boulanger et al. 2006]. Techniques such as Delaunay triangulation have been widely studied. They can be used to triangulate the area inside or around the polygon-shaped obstacle, so that triangles follow the boundaries of obstacles [Boulanger et al. 2006; Buyukkaya and Abdallah 2008] as shown in the figure.

Games can also be divided into *mini worlds* that have a free format and are connected to each other such as countries in the game world, with portals for moving between them [Chen et al. 2005a; Knutsson et al. 2004] as in Figure 2(a). This is particularly useful when AOI always resides within a single zone. While mini worlds do not offer the concept of a continuous world, other approaches do. The zoning is only virtual, done merely for the purpose of interest management, and is not visible to the players.

Determining the right size for zones is challenging. Different games have very different characteristics, and even inside a game, different parts may require different zoning mechanisms. Therefore, an optimal zoning mechanism for all cases does not exist. A very large zone can result in too many objects in a single zone, making interest management less efficient as any player might only be interested in a small set of these objects. On the other hand, too small a zone may be much smaller than the AOI of players, resulting again in complex interest management between multiple zones. This trade-off is addressed by dynamic interest management schemes.

Limits of AOI and Dynamic Zoning. AOI filtering suffers specific user behaviors, such as *flocking*, which refers to the movement of many players to one area in the game world [Pittman and GauthierDickey 2007; Chen et al. 2005a]. This can adversely affect replication management (see Section 7). Game hotspots form since some areas become more interesting or more profitable to the players in terms of experience points, treasures, new quests, or invitations by other players. This results in a large number of players coming to the same area while other areas become less populated. Hotspots can change quickly as players move to new interesting areas as they emerge, making it often difficult to prepare for such changes in advance. Populations in real games often follow a power law distribution [Pittman and GauthierDickey 2007], and the increase in the number of players in the area can result in a quadratic increase in the network traffic generated due to the increase in the number of interactions between players [Bharambe et al. 2008]. Varvello et al. [2011] find that in *Second Life* the number of objects per region is roughly constant over a one month period and that the active population at any point of time is between 30,000 and 50,000 avatars, that is, about 0.3% of the registered avatars. About 30% of the regions are never visited in a six day period and less than 1% of the regions have large peak populations. Avatars tend to organize in small groups of 2 to 10 avatars. Large groups of avatars are very rare and are driven by the presence of events such as concerts and shows. As a result, game designers may have to use artificial means to discourage players from gathering in the same region. This can prevent certain types of interesting game play such as *epic battles* [Blizzard 2011]. A more comprehensive study of player movements in Massively Multiplayer Online Role Playing Games (MMORPG) and session patterns is provided in Varvello et al. [2011], Suznjevic et al. [2009], and Miller and Crowcroft [2010].

Flocking can be partially addressed with dynamic zones. As more players move into a popular zone, the zone can be split. However, if zones become significantly smaller than the typical AOI, splitting does not really help anymore as inter-zone communications become the bottleneck. One possibility then is to shrink the AOI of the players. However, this might lower the game experience for the players. In addition, depending on the game design, AOI can be further broken down into different types [Morse 1996] but for the purpose of this article we use the general term AOI.

2.8. Consistency Control

In a distributed architecture like a multiplayer game, concurrent and possibly conflicting updates may be executed at different sites resulting in inconsistent states. Inconsistencies occur due to the execution of parallel and conflicting updates, and consistency mechanisms have to *avoid* or *correct* them. For example, if two players shoot a third player, nearly at the same time, all players should see the updates in the same order and only the first shot should be successful at all replicas.

Generally, systems are built such that if all replicas execute all updates in the same order, then all sites will have the same state. However, if messages are received out of order, their execution can result in inconsistent state if the out-of-order messages are causally dependent. For example, if a player drinks a healing potion to increase his health points and then, and only based on the increased amount of health, is able to pick up a sword, these actions have to be performed in the same order at all replicas, otherwise inconsistencies might become visible. Other types of inconsistency are caused by loss of updates. Most games, in order to deal with latency issues, use the fast but unreliable UDP messaging protocol where message loss is possible. Solutions are to send some updates with reliable TCP, or to use commit protocols for critical actions, in particular if they change the state of several objects and atomicity is required [Mauve et al. 2004; Bharambe et al. 2008, 2006; Chandler and Finney 2005].

Consistency Definitions. Many definitions for consistency exist and consistency control has been studied in other distributed contexts (e.g., shared memory [Adve and Gharachorloo 1996] or distributed systems [Özsu and Valduriez 2011]). A very strong form of consistency is achieved if every interaction is treated as a transaction that provides the transactional properties such as isolation and atomicity. However, providing transactional properties is often costly and might not be feasible for all game actions. For instance, running a two-phase commit protocol to guarantee atomicity leads to large latencies which could reduce the interactivity of the game significantly. Eventual consistency [Terry et al. 1995] is a weaker form of consistency. It allows individual copies of an object to be temporarily inconsistent but eventually consistent, meaning that if update activity did cease for sufficiently long time all object copies would eventually have the same state. Generally, there exist well-known trade-offs between performance and consistency restrictions, which can also be applied to MMOGs. This fact has been well presented in Brewer's "CAP conjecture" [Gilbert and Lynch 2002], indicating that a system can only achieve two out of three properties of consistency, availability, and handling network partitions. As interactivity is of essence to the success of most games, they often sacrifice consistency for other goals, and as a result, games usually provide inconsistency resolution instead of inconsistency prevention, enforcing not more than eventual consistency [Chandler and Finney 2005; Mauve et al. 2004].

Different levels of consistency, implemented by various mechanisms, might be considered for different object and interaction types as not all interactions are of the same importance [Zhang and Kemme 2011]. For example, many virtual game objects are considered valuable and can be sold or traded for real money, making consistency control a critical requirement. In 2009, micro-transactions generated 250 million dollars in U.S. alone, with the most expensive item being 635 thousand dollars[3], and are projected to hit 13.6 billion dollars by 2014 worldwide[4]. In contrast, player position updates have much lower consistency requirements, and eventual consistency will be sufficient.

[3] Planet Calypso Player Sells Virtual Resort for $635,000 USD http://www.prnewswire.com/news-releases/planet-calypso-player-sells-virtual-resort-for-63500000-usd-107426428.html.

[4] Magid Associates and PlaySpan Release 2nd Annual Survey on Virtual Goods Market Penetration and Growth in North America. https://developer.playspan.com/developer/pdf/PlaySpan_Magid_5_27_10_Final.pdf.

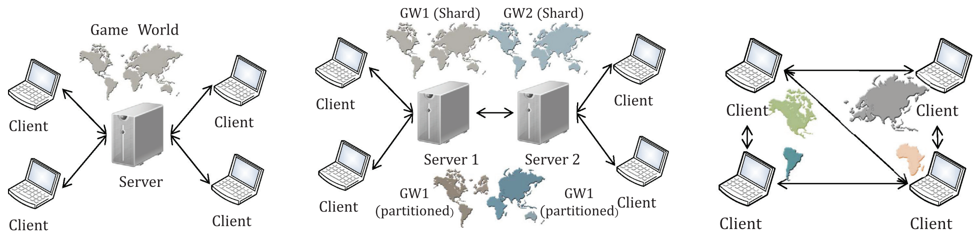
Stale Views. In the previous sections we have introduced the primary copy replication model that requires that all updates are first performed at the primary copy. This simplifies consistency management as it is not possible to have conflicting updates performed at different copies. Instead all updates are serialized at the primary copy. However, replicas will receive the update changes only some time after they occur at the primary. During this period, replicas are *stale*. Players observe these state values and might initiate invalid update requests to the primary copy, based on these stale values. Thus, actions submitted by players based on their local state might lead to results that the players did not anticipate. Ideally, the primary copy directly sends updates to all replicas in order to minimize staleness. However, this is expensive and other methods may be employed that may increase the latency and staleness experienced by the replicas. For instance, in *Second Life*, in a region crowded with virtual objects, avatars have an inconsistent view of their neighbor avatars half of the time, meaning either they do not see them or they see them at a wrong location. Moreover, in 50% of the cases, this inconsistency lasts more than one second [Varvello et al. 2011].

Consistency Techniques. Games use several techniques to hide staleness and inconsistencies due to the latency of update propagation. These techniques can also be used to hide message loss. They typically fall under the two categories of *Predictive Contract Mechanisms (PCM)* [IEEE 1995] and *multiresolution simulation* [Hamilton et al. 1997] and are often used together. Dead reckoning is a common form of PCM [Pantel and Wolf 2002a; Gautier et al. 1999]. It was originally designed in the nautical and aviation domain to calculate the current position of the plane based on the previous position and the motion vector. It is used in games to calculate the position of a player in the upcoming frame based on her previous location and her speed and direction of movement. It can also be used to detect collisions that will happen in the future. Dead reckoning is used if messages do not arrive in time because of delays or loss. However, it is also possible that all replicas continuously perform dead reckoning, and the primary copy only sends a state update to the replicas if it detects that the state calculated by dead reckoning has a higher difference from the true state than a certain threshold. Neural networks have been proposed as predictors [McCoy et al. 2007] for PCMs as well as a hybrid dead-reckoning/shortest-path predictor [McCoy et al. 2005]. AntReckoning [Yahyavi et al. 2011, 2012] uses ant-colony-inspired methods to model players' interest and improve the accuracy of the predictions based on a recent history of players' movements and attraction or repulsion of surrounding game items.

Measuring Consistency. The degree of inconsistency can be defined by comparing the (potentially inconsistent) state at each client to a virtual perfect site that receives and executes all interactions with no delay and in the right order [Mauve et al. 2004]. Chen and El Zarki [2011] provide an objective evaluation framework for Quality of Experience (QoE) in games. QoE takes three basic perceptions: responsiveness, precision, and fairness. *Responsiveness* is the time the system takes to respond to an event, *precision* is the degree of accuracy required to complete an action successfully, and *fairness* is the degree of difference among all players' gaming environments. Similarly, Varvello et al. [2011] consider inconsistency, interactivity, and discovery latency to measure QoE.

3. MMOG ARCHITECTURES

The main game architectures for MMOG, as shown in Figure 3, are the traditional client-server architecture, MultiServer architectures (MS), and peer-to-peer (P2P) architectures. Different architectures try to achieve scalability through various means. Scalability can be achieved either by: (1) increasing the resources or by (2) reducing the consumption. The methods discussed before such as AOI filtering and dead reckoning are designed to reduce the resource consumption. Server-based architectures try to



(a) Client-Server (b) Multi-Server (c) P2P

Fig. 3. Different gaming architectures: (a) In a client-server architecture the server is responsible for maintaining the whole game world; (b) in a multiserver architecture either: (1) servers maintain completely separate game worlds (*shards*) or (2) the game world is divided into different zones maintained by different servers; (c) in a peer-to-peer architecture each peer maintains a part of the game world.

increase the amount of resources by adding multiple servers to distribute the load. P2P architectures increase the resources by using the resources available to each client as they join.

3.1. Client-Server Architecture

The typical client-server architecture is shown in Figure 3(a). In this architecture, the server holds the master copies of all mutable objects and avatars and maintains global knowledge of the game world. Clients connect to the server to receive the necessary information about the game world. All player updates and player interactions are sent to the server for execution as well as conflict resolution, and the server is responsible for sending object updates to all interested players after the updates have been executed. The main drawback is that even the best-provisioned servers can only support a limited number of players. The common solution is to add multiple servers to improve scalability.

3.2. Distributed MultiServer Architecture

The multiserver architecture is shown in Figure 3(b). Large game companies usually maintain server farms to provide service for the clients [Kulkarni et al. 2010; Terazona 2002; Butterfly.net 2003]. Multiserver architectures can be divided into two categories. In the first category, several complete instances of the game world exist, also called *shards*, and each shard is maintained by a server. Every server is responsible for a different set of clients and has a complete copy of the his own game world. That is, each set of clients and their server follow the traditional client-server architecture. There is usually no need for communication between these servers. Games and maps should be designed in a way that artificially prohibits players from moving between different game instances. Users are typically assigned to servers based on their geographical location and each server is responsible for a separate region (e.g., North America, Europe, etc.).

In the second category, only a single game world exists that is divided into several regions. Each region is maintained by a separate server. Players are all in the same game world; however, in most games they are only able to interact with other players in the same region. Players are allowed to move between regions but this requires support for a hand-off mechanism between servers which can be transparent to the player. As the player moves near the border of a region the necessary information by the neighboring region is sent to the player and the player can easily cross the border. The hand-off mechanism also may not be transparent as the player is asked to go through a special portal or gateway in order to enter a new region. Thus, the concept of regions can be used for interest management, as described in the previous section, as well as for load balancing among servers.

3.3. Peer-to-Peer (P2P) Architecture

Another option to increase scalability is the use of a peer-to-peer architecture as shown in Figure 3(c). Basically every node acts both as a server and a client. In the context of games, this means that each node could become responsible for maintaining master copies of some of the game objects and/or for update dissemination to other nodes. This model can be highly scalable since the load is distributed among all nodes and addition of new nodes also introduces new resources to the system.

3.4. Characteristics and Comparison

Here we discuss advantages and disadvantages of each of these architectures.

Server Architecture. Centralized architectures provide the highest level of control over the game world. The game state can be the most valuable property of game developers, especially in case the game world is persistent. Game creators can easily change and update the game state and have control over the necessary updates in the software such as software patches, character updates, addition of new missions and expansion packs. *Manageability* and *control* are two main reasons why game companies typically use this architecture. Another reason is that programming in server-based architectures is simpler in comparison to P2P architectures. Overall, the client-server architecture is the simplest of these architectures.

Easy consistency management is another reason for the popularity of centralized architectures. Since the server executes all updates and resolves conflicts, managing consistency is simpler than in multiserver architectures requiring hand-offs, or P2P architectures that have even higher distribution.

The biggest drawback of a single-server architecture is scalability. Even the best-provisioned servers are not able to handle more than a few thousand players. This number is further reduced for games that have stringent latency requirements. For example, the Quake II game (a popular FPS MOG) uses a client-server architecture and its scalability in the traditional client-server architecture is only a few hundred as measured in Bharambe et al. [2006].

Further problems are cost and fault tolerance. A single server that is able to handle the massive load will be extremely expensive, and it remains a single point of failure. Its failure will interrupt game play and nonpersistent game state will be lost. This problem can be solved by adding backup servers that can take over in case of failure. But maintaining backup servers introduces more complexity and cost, and may result in even further decrease in scalability.

Multiserver Architectures. Maintaining multiple servers improves scalability and inherits other benefits of the client-server architectures. Since the game world is divided into instances or regions and each instance is maintained by a separate server, a higher number of players can be supported simultaneously. As servers can serve as backup for other servers, this scheme has also the potential for higher fault tolerance than a single-server system. But even if no backup mechanism is used, the failure of one server will only affect the players connected to this server and not interrupt execution at other servers. *Second Life* contains 18,000 regions where each of the regions is managed by a dedicated server [Varvello et al. 2009b].

One of the main drawbacks of multiserver systems is isolation of players if shards are used. Players can only interact with other players in the same instance of the game world and are not able to interact with players in other instances. If the game supports movements between regions, a complicated hand-off mechanism [Knutsson et al. 2004] is required that maintains game state consistency. Furthermore, a single region hosted by a server can only support a limited number of players. In case of flocking behavior, even the best-provisioned servers might not be able to provide service for an overloaded region. In addition, multiserver systems cannot be scaled without limit if a region-based approach is used, as the game world cannot be divided infinitely into smaller and smaller regions. Most game developers address this issue by expanding or adding new maps as the number of players increases in order to maintain a low population density. However, this approach prevents the players from experiencing a more interactive environment. Another issue with many regions is the inter-server communication due to player movements. Different methods have been proposed in order to balance the load between regions. Most architectures assume that a single server is able to handle the load on a single region and if the load on a region is low, a server may maintain several regions, preferably adjacent to each other. Dynamic load-balancing schemes [Chen et al. 2005a] have been proposed to dynamically allocate overloaded regions to new servers while maintaining region locality in games. A combination of both shards and regions can also be used by game companies.

Finally, a major drawback is the cost. Maintaining server farms is very expensive and can prevent start-up or small game companies from entering the MMOG market. Acquiring servers for 30,000 simultaneous players can be about $800,000 and the bandwidth costs can reach hundreds of thousands of dollars [Mulligan et al. 2003]. Maintenance costs such as cooling can match these numbers.

P2P Architectures. In principle, P2P architectures have the highest potential for scalability as every peer that joins the game adds new resources to the system. A particular advantage is that these resources are added at no extra cost for the game provider. As the load is managed by the peers the need for expensive servers is greatly reduced if not eliminated. Furthermore, as responsibility is distributed across many nodes, each of them carrying only a bit of the load, the failure of any individual peer should not affect many other players if churn is handled appropriately. Moreover, as will be discussed in more detail in Section 5, if direct connections between peers are created properly, low latency can be achieved as there is no need to send the updates to a central server and from the server to other clients. Instead, updates are directly sent to interested peers. For example, P2P *Second Life* improves user experience, measured in terms of consistency, by about 20% compared to a client-server architecture, and avatar interactivity is also five times faster [Varvello et al. 2009b].

On the other hand, P2P architectures have several drawbacks. One of the major issues with P2P architectures is security. Cheating is easier in a P2P environment [Baughman et al. 2007; Kabus et al. 2005; Goodman and Verbrugge 2008]. Another problem is that P2P systems are harder to manage and control given that there is no central server that maintains global knowledge. Since the state is distributed among peers, it would be hard for game companies to have complete control over the game. Providing consistency control in P2P systems is also more difficult since conflicting updates might be executed at different sites, resulting in inconsistency. Generally, coordination overhead and complexity is likely to rise with the number of nodes in the system.

A summary of the advantages and disadvantages of each architecture is provided in Table I.

3.5. Hybrid Architectures

It is possible to combine a P2P architecture with a server-based architecture. These approaches can be divided into several categories according to what is being handled by the P2P system (see Table II).

Table I. Comparison of Different Architectures

Architecture	Pros	Cons
Client-Server	+ Simplicity + Easy management + Consistency control	−− Scalability −− Fault tolerance −− Cost
Multi-Server	+ Scalability + Fault tolerance	− Isolation of players − Complexity −− Cost
Peer-to-Peer	++ Scalability ++ Cost + Fault tolerance	− Harder to develop − Consistency control − Cheating

Cooperative Message Dissemination. The game state is maintained by one or multiple servers but update dissemination uses a P2P approach. Typically, players send their interactions directly to the server. After the execution, the server uses a P2P multicasting mechanism to send the updates to the players. This allows for a reduction in bandwidth requirements (and cost savings) at the server side.

State Distribution. The game state is distributed among peers. Peers can hold primary copies of objects and thus, are responsible for execution of player actions. However, all or part of the communication between peers may be managed by the servers. Moreover, the servers are responsible for some centralized operations such as authentication, and keeping track of joins and leaves of players. These architectures achieve scalability by distributing the cost of state execution among clients.

Basic Server Control. Both message dissemination and state distribution are done only through the P2P overlay. The servers' primary role is to keep highly sensitive data, such as user logins, payment information, and players' progress and state. They may perform authentication and admission control, meaning the server controls joins and leaves. Servers may also coordinate some types of interactions between the peers, for example, those that require the highest consistency. However, they do not maintain games state or perform state dissemination.

P2P systems can be of other uses to game developers as well. In *World of Warcraft*, for example, P2P systems are used to distribute software updates to clients. These systems can be similar to popular P2P file distribution systems. In this article, however, we focus on P2P systems that are used for maintaining game state or to facilitate update dissemination.

3.6. Heterogeneous and Homogeneous P2P Systems

In *heterogeneous* solutions, some of the peers behave as *superpeers* (also referred to as coordinators [Knutsson et al. 2004], responsible nodes [Yamamoto et al. 2005], and master nodes [Yu and Vuong 2005]) and some as *normal* nodes. Superpeers are responsible for most functionalities of the architecture. Superpeers can be chosen among the normal nodes that are given higher responsibilities or can be nodes with higher levels of resources (such as federated servers). Most architectures that divide the game world into regions and have a coordinator for that region deploy superpeers.

In *homogeneous* approaches, all nodes have similar responsibilities. For example, in pSense [Schmieg et al. 2008], even though there are several types of possible roles for nodes, such as sensor nodes and neighbor nodes, all nodes have these responsibilities, resulting in a homogeneous architecture. Most mutual notification systems fall under this category.

Table II. Comparison of Representatives of Different P2P Architectures (N/A: not applicable)

Architecture	Type	Network Overlay	Interest Management	Replication & Consistency Control
SimMud [Knutsson et al. 2004]	Structured	DHT: Pastry + AL Multicast: Scribe	Static Regions: Rectangular Cells	Primary Copy: Region Controller
N-Trees [GauthierDickey et al. 2005]	Structured	DHT: Pastry + AL Multicast: N-trees	Nested Regions (scopes) + Scoped events: Grid Cells	Primary Copy: Region Controller
Colyseus [Bharambe et al. 2006]	Structured	DHT: Mercury + Direct	Dynamic Rectangular Cells + Prefetching objects	Primary Copy: Each Peer + Proactive Replication + Soft-State Storage
P2P Second Life [Varvello et al. 2009b]	Structured	DHT (Kad)	Dynamic: Adaptive Cells	Primary Copy: Region Controller
Badumna [Kulkarni et al. 2010]	Structured	DHT	Regions + Bounded dynamic AOI	Primary Copy: Region Controller
pSense [Schmieg et al. 2008]	Unstructured	Direct(Neighborhood) + Automatic forwarding	Dynamic Circular AOI	N/A
ASCEND [Hu and Liao 2004]	Unstructured	Direct(Neighborhood) + Connectivity assurance	Dynamic AOI, Voronoi Diagrams	Primary Copy: Each Peer
Solipsis('03) [Keller and Simon 2003]	Unstructured	Direct(Neighborhood)	Dynamic AOI, Convex Hulls	Primary Copy: Each Peer
Solipsis('08) [Frey et al. 2008]	Unstructured	Direct(Neighborhood) based on RayNet	Dynamic: Voronoi	Primary Copy: Each Peer
Message Exchange Scheme [Kawahara et al. 2004]	Unstructured	Direct(Neighborhood)	Dynamic BW dependent AOI, Neighborhood	Primary Copy: Region Controller
Donnybrook [Bharambe et al. 2008]	Unstructured	Direct (All-to-All) + Forwarding pools	Game world, Interest Sets	Primary Copy: Each Peers + Doppleganger
Amaze [Berglund and Cheriton 1985]	Unstructured	Direct(All-to-All)	Game world	Primary Copy: Each Peer
VSM [Hu et al. 2008]	Unstructured	Direct(Neighborhood)	Dynamic AOI, Voronoi diagram	Primary Copy: Each Peer
MiMaze [Diot and Gautier 1999]	Hybrid: Server + IP Multicast	IP Multicast Groups	Game World	Primary Copy: Each peer
FreeMMG [Cecin et al. 2004]	Hybrid: Server + Unstructured	Server + Direct Communication	Static Regions: Rectangular Cells	Primary Copy: Each Peer
Quazal [Quazal 2011]	Hybrid: Server + Unstructured	Server + Direct Communication	Duplication Spaces	Duplication Spaces

Table II. Continued

Architecture	Type	Network Overlay	Interest Management	Replication & Consistency Control
Dist. Avatar Mgmt. [Varvello et al. 2009b]	Hybrid: Server + Unstructured	Direct(Neighborhood)	Dynamic: Region + Delaunay Triangulation	N/A
Hydra [Chan et al. 2007]	Hybrid: Server + Unstructured	Region Server + Proxies	Static Region	Primary Copy: Region(slice) Controller
Distributed Event Delivery System [Yamamoto et al. 2005]	Hybrid: Server + Structured	Server + Load balancing tree	Static Regions: Rectangular Cells	Primary Copy: Region Controller
IRS [Goodman and Verbrugge 2008]	Hybrid: Server + Structured	Server Message Relaying + Proxy Execution	Server	Primary Copy: Proxy peer
MM-VISA [Ahmed et al. 2009]	Hybrid: Server + Structured	AL Multicast	Hexagonal Regions + Player Clusters	Primary Copy: Region Controller
MOPAR [Yu and Vuong 2005]	Hybrid: Structured + Unstructured	DHT: Pastry + AL Multicast: Scribe	Hierarchical AOI + Static Regions: Hexagonal cells	Primary Copy: Region Controller
VoroGame [Buyukkaya et al. 2009]	Hybrid: Unstructured + Structured	Direct(Neighborhood) + DHT	Dynamic: Voronoi (Convex Polygon)	Primary Copy: Each Peer (Random DHT)
Mammoth [Kienzle et al. 2009]	Hybrid: Server + Unstructured	Region Controller(All-to-All)	Dynamic Regions: Rectangular tiles	Primary Copy: Region Controller
Zoned Federation [Iimura et al. 2004]	Hybrid: Server + Structured	DHT	Static Regions: Rectangular Cells	Primary Copy: Region Controller

3.7. Other Functionalities

Inter-server communication and streaming. Multiserver environments can also benefit from many of the techniques presented in P2P architectures. The idea is to deploy peer-to-peer protocols for the communication among servers [Kulkarni et al. 2010; Iimura et al. 2004; Bharambe et al. 2006]. Servers are treated as peers and the same state distribution and update dissemination mechanisms suggested for a peer-to-peer architecture are applied to the server system. Slight modifications will be necessary in order to differentiate between servers (peers in this case) and game clients. This is further discussed in Section 10.

4. P2P ARCHITECTURES

Here we discuss different P2P architectures proposed and categorize them according to the underlying P2P structure.

4.1. Introduction

Peer-to-peer systems are usually created by ad hoc addition of nodes and build an application-layer overlay network on top of the network layer. A fundamental requirement for very high scalability is that no single peer can know all other peers in the system. Instead it is always only “connected” to a fraction of peers. A prominent use of P2P systems has been in content (file) distribution systems. Instead of having a single server providing content to all clients, peers can download content from other peers, and in turn offer their content to others. In order for a peer to find the content it is looking for, efficient query routing protocols have to be in place.

Based on how nodes are connected to each other, P2P overlays are generally divided into two types: *structured* and *unstructured*. In structured P2P systems, a deterministic protocol is used to form a specific graph structure (by deciding which peers build connections to each other) that ensures that any node can route a message to any other node (or find an object) in the network overlay (by exchanging $O(\log(N))$ messages, where N is the number of nodes). Key-based routing mechanisms are used to provide a lookup service similar to a hash table but different in that the (key, value) pairs are distributed throughout the nodes. They are able to add or remove nodes from the overlay with low overhead. Examples of such P2P substrates are Pastry [Rowstron and Druschel 2001a], Chord [Stoica et al. 2001], Tapestry [Zhao et al. 2001], and Mercury [Bharambe et al. 2004]. Distributed Hash Tables (DHTs) can use these substrates as the underlying mechanism for providing a hash table functionality distributed over many nodes.

In unstructured P2P systems, no deterministic algorithm for organization and optimization of network connections between peers exists. Network connections are generally established randomly or using probabilistic mechanisms that aim to converge into a suitable overlay (e.g., by connecting with a higher probability to nodes that are semantically close). Search in such networks is often done in a probabilistic manner, and in order to speed up the search, redundancy mechanisms, such as flooding or content replication, are used [Sozio et al. 2008; Nagel et al. 2006; Costa et al. 2003]. Gnutella and Freenet are examples of unstructured P2P systems. The basic idea of many P2P-based games is to distribute the game state among peers and along with it processing, network, and storage tasks. In a primary-copy-based replication scheme, this means distributing primary copies of objects among peers. Furthermore, the P2P overlay can be used for update dissemination. Both structured and unstructured P2P gaming architectures have been proposed. A summary of some of the proposed P2P-based gaming solutions is presented in Table II[5]. This table is not meant to include all the P2P architectures that have been developed for games but to compare examples that exploit different architectures and strategies. We will describe many of these systems in this article.

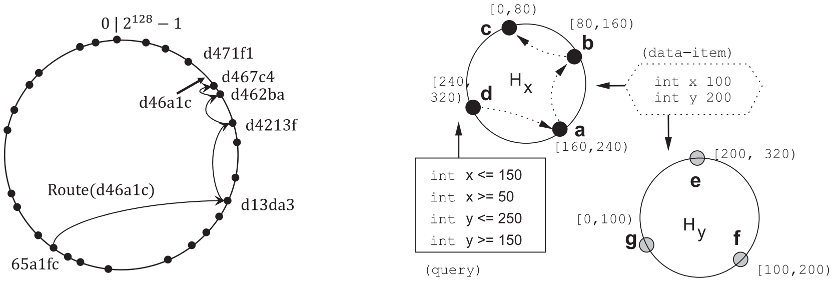
4.2. Structured P2P Game Architectures

Structured P2P architectures typically use a DHT as the underlying mechanism for game state distribution and update dissemination. In this article, we will describe SimMud [Knutsson et al. 2004] in detail as an example of a DHT-based system. Most other solutions use a conceptually similar approach but optimize various aspects of the architecture.

SimMud uses Pastry [Rowstron and Druschel 2001a] to distribute game objects and Scribe [Castro et al. 2002b] for update dissemination. Here we first shortly describe these protocols and then explain SimMud’s use of them.

Figure 4(a) shows Pastry’s overlay and message routing. In Pastry, every node has a 128-bit NodeID. NodeIDs can be uniformly distributed using a hash code (e.g. SHA-1) of the nodes’ IP addresses or their public keys. In general, in order to distribute objects

[5] For an alternative list see: http://vast.sourceforge.net/relatedwork.php. While the categorization and terminology is somewhat different from this article, it provides a comprehensive list of related work.



(a) Pastry Message Routing

(b) Attribute Hubs in Mercury

Fig. 4. (a) Pastry [Castro et al. 2002a] routing (with permission of Castro et al.). In each jump the message with a given key is routed to a node with a digit more in common with the key. The keys are associated with game objects in SimMud [Knutsson et al. 2004]; (b) mercury network overlay [Bharambe et al. 2004] (©ACM 2004). A different attribute hub is created for each attribute. A node in each hub is responsible for a range of that attribute. Data items are assigned to attribute hubs based on their attribute values. Players in Colyseus [Bharambe et al. 2006] use the range queries that specify their AOI for object discovery in games.

using Pastry (here object refers to any application object), each object is given an ObjectID (a hash code of the object) and its primary copy is placed on the Pastry node whose NodeID is closest to the ObjectID. An object can later be looked up by routing a message with ObjectID as the key. The message will automatically be routed to the node that has the object. Thus, when a node with a replica wants to update an object, it has to send the update message with the ObjectID as key and it will be automatically received by the node with the primary copy, which can then perform the update. The design goal in Pastry is the ability to efficiently route a message with a given key to the node with the closest ID to that key. If the number of nodes in the network is N, Pastry routes a message in $\lceil \log_{2^b}(N) \rceil$ steps on average (b is a configuration parameter) using a routing table with $\lceil \log_{2^b}(N) \rceil \times (2^b - 1)$ entries. Pastry also provides the means for efficient node addition and deletion and can handle node failures.

Scribe [Castro et al. 2002b] is a multicast primitive built on top of Pastry. For each multicast group Scribe builds a multicast tree with nodes being the peers of the overlay. Nodes can join and leave multicast groups. Any multicast message to a group is sent to all current members of the group, where it is sent to the root node of the group and is then forwarded along the multicast tree structure. The idea is that the root node is the node whose nodeID is closest to the GroupID. Therefore, to find the root node, Scribe uses the query routing protocol of Pastry. Scribe provides best-effort delivery of multicast messages and is fully decentralized.

SimMud follows the approach of traditional multiserver architectures by dividing the game world into several regions (similar to Figure 2(a) where the region is AOI). In SimMud, game objects always reside in a single region and a player is only able to see game objects of the same region. To exploit Pastry, each region builds a group with a RegionID (a hash of the region's name), and the peer with the NodeID closest to the RegionID becomes the coordinator of the region. Objects in a region are grouped together and are mapped to the coordinator in the Pastry key space. That is, the coordinator of the region has the primary copies of all objects residing in the region. When a new player enters the region it searches for the coordinator and receives replicas of all objects in the region. All updates on game objects of the region are sent to the coordinator, who serializes them, resolves conflicts, and sends state updates to interested players.

For update dissemination, Scribe is used. SimMud creates a multicast group for each region such that its GroupID equals the RegionID. This means that the coordinator of the region becomes the root of the multicast tree and simply sends updates along the multicast tree. All players of the region subscribe to the multicast group and receive the updates.

In SimMud, regions are static and each region is assigned to a coordinator via the Pastry's single-attribute DHT. The Colyseus game architecture [Bharambe et al. 2006] offers a more dynamic mechanism by exploiting Mercury [Bharambe et al. 2004], a DHT substrate that supports range queries on several attributes (Figure 4(b)). For every attribute, a number of nodes are joined together in a group called a hub to answer queries about this attribute. The range of an attribute is broken down into several smaller ranges. Every node in the attribute hub becomes responsible for one of these smaller ranges. Similarly, Tanin et al. [2007] use quadtree index structures to support more complex queries such as range queries on various data types.

The main idea of Colyseus is to have multiple attributes, one for each dimension of the game world. A peer being responsible for a range of the x or y attribute knows about all game objects in this range of the game world. A player interested in game objects within a certain range searches for the peers that are responsible for this range via the Mercury query routing mechanism. These peers then inform the player about all objects that reside within the specified range. Colyseus uses Mercury only as a mechanism for discovering objects in the area of interest and receives updates afterwards directly from the holder of the primary copy. Objects periodically publish their updated attributes to Mercury to keep it up to date.

Other structured architectures such as GauthierDickey et al. [2005] rely on the same principles presented here. However, they might use different DHTs, interest management, or publish-subscribe systems. Similarly, P2P *Second Life* [Varvello et al. 2009a] provides a region-based structured system, however, it uses *Kad*[6] as the underlying substrate.

4.3. Unstructured P2P Game Architectures

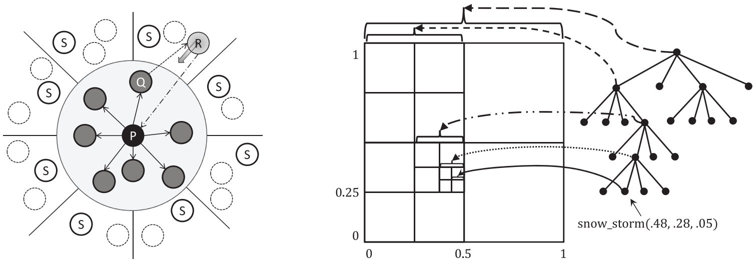
Here we loosely define an unstructured architecture as one that lacks a global mechanism such as a DHT that would manage joins and leaves, maintain the overlay, or provide global routing protocols.

Most proposed unstructured P2P architectures are based on a *mutual notification system* [Chu et al. 2002]. In these systems it is not necessary to divide the world into regions; instead, players use a mechanism to detect other players in their AOI and directly send their updates to these players. In order to detect players that move into the AOI of a player, players depend on notifications of their neighboring players. Here, we explain pSense [Schmieg et al. 2008] as an example of such systems.

pSense is designed to manage position updates of the players and does not take other action types into account. Its design is based on the fact that peers in the AOI of a player should receive updates rapidly while other peers do not need to receive updates at all. Therefore, whenever a player moves, updates are sent to all nodes in the AOI of the player. In order to do this, a list of the nodes in the AOI is maintained, called the neighbor list. Whenever a node leaves the AOI, it is removed from the list.

In order to detect the nodes that enter its AOI, a node p relies on its neighbors, that is players close to this player in the game world (Figure 5(a)). If a neighbor q receives an update of p and knows about an approaching node r (which is already in the AOI of q and will soon enter the AOI of p) it forwards the update to r. Thus, r gets to know p and will send its own updates to p in the future. In order to improve on this system, each node p also maintains a set of sensor nodes outside of its AOI as shown in Figure 5(a). Node p also sends its updates to its sensor nodes which can then forward

[6] http://www.emule.com.



(a) pSense neighbor discovery

(b) N-trees and scopes

Fig. 5. (a) pSense neighbor discovery and sensor nodes[Schmieg et al. 2008] (adapted, ©IEEE 2008). A neighbor forwards a position update to an incoming node which in turn forwards its own update to the node resulting in a new neighbor discovery; (b) the game world is divided into nested regions represented by a tree where leaf nodes each represent a region and the parent node represents the sum of all its children regions. An event (such as a snow storm) is only sent to peers that are in regions overlapping with the event's scope and only travels upwards when even scope is bigger than the current region [GauthierDickey et al. 2005] (adapted, ©ACM 2005).

them to nodes that are entering *p*'s AOI. Sensor nodes also help to prevent the overlay from becoming disconnected (see Section 6).

In order to determine the sensor nodes around the local node, several approaches can be used. One option is to divide the game world around the local player into *n* equally sized angles. The nearest node in each angle that resides outside the vision range is chosen as the sensor node. In order to maintain the sensor node list, when a sensor node receives a request, it can suggest other nodes that it thinks are better suited. This new node replaces the old sensor node.

Several other mutual notification systems [Hu and Liao 2004; Kawahara et al. 2004; Keller and Simon 2003; Jelasity et al. 2009] exist that use mechanisms similar to pSense to build an AOI-aware overlay but use different neighbor discovery mechanisms as will be discussed in Section 6. While pSense exploits update dissemination to maintain neighbor lists, other mechanisms can also use detection mechanisms that regularly exchange neighbor lists, typically through gossip messages.

All-to-all is another form of unstructured P2P architecture where all players maintain the full game state and use point-to-point communication for update dissemination. An example is *Amaze* [Berglund and Cheriton 1985], developed in 1985, or *Donnybrook* [Bharambe et al. 2008]. In such systems, special methods are needed to reduce bandwidth requirements. We will discuss these mechanisms in Section 5.

4.4. Hybrid P2P Architectures

Several approaches rely on a combination of the architectures discussed so far. Many of the solutions discussed, such as SimMud and Colyseus, can be used in combination with a server as well as in a completely serverless setup. As in Section 3.5, we refer to solutions that use a combination of different P2P architectures or a combination of servers and P2P architectures as a hybrid architecture.

Hybrid client-server and P2P architectures. In FreeMMG [Cecin et al. 2004], the server is responsible for maintaining user-sensitive data such as user names and passwords. But it also divides the game world into segments and performs interest management. Similar functionalities are provided by the servers in Yamamoto et al. [2005]. In Quazal [2011] servers only do user management. In IRS [Goodman and Verbrugge 2008] the game state execution is distributed between peers but, contrary to many architectures, the server is responsible for message dissemination between peers and running security checks on the updates.

In MM-VISA [Ahmed and Shirmohammadi 2010; Ahmed et al. 2009], the authors propose a fault-tolerance procedure for a hybrid coordinator-based system. They focus on the difference between slow-moving players, for example, a soldier, and a fast-moving player, such as a tank, in the game world. A fast-moving player may move to other regions too fast, breaking all the necessary connections between the players before they can be repaired. They propose the use of player clustering, based on the type of their activities, inside each region to deal with fast-moving players. Players in clusters form their overlay according to the cluster they belong to. Therefore, clusters isolate different types of players in the structure: a leaving player only affects the dependent nodes of that cluster, not the whole zone.

In Najaran and Krasic [2010], the authors propose a hybrid scheme where interest management is performed in the cloud. It relies on P2P multicast trees, such as Scribe, to perform pub-sub functionalities.

Hybrid P2P-P2P architectures. In contrast to the aforesaid systems that combine P2P solutions with a server, MOPAR [Yu and Vuong 2005] is a special form of hybrid system as it combines two types of P2P architectures: structured and unstructured. MOPAR is similar to SimMud in its design and divides the game world into zones controlled by coordinators, and uses Pastry and Scribe for state distribution and update dissemination to the peers. However, coordinators also periodically exchange their neighborhood information. While this is conceptually similar to an unstructured message gossip, MOPAR does this exchange in a hierarchical manner. Coordinators only exchange their neighborhood information with the coordinators of neighboring zones and watch the neighbors for peers that might have subscribed to them. VoroGame [Buyukkaya et al. 2009] uses a combination of structured and unstructured systems as follows: the structured system is based on a DHT and is used to maintain game objects. However, instead of using fixed regions, a mutual notification system based on Voronoi diagrams is used for neighbor discovery, providing more dynamic interest management (see Section 6). While these approaches benefit from the advantages of both architectures, one major drawback is the cost of maintaining two separate overlays, in terms of network bandwidth, memory overhead, and processing power.

Note that many of the systems discussed can be run in both hybrid and pure peer-to-peer fashion. Most coordinator-based systems fall in this category. VSM [Hu et al. 2008], on the other hand, is an example of an unstructured Voronoi-based mutual notification system that uses superpeers, be they peers with sufficient resources or servers, for state management. The neighbor discovery in this approach is similar to other unstructured Voronoi-based architectures as discussed in Section 6.2. However, in this case the game state is managed by superpeers or *arbitrators* that can be servers.

5. COMMUNICATIONS & MULTICAST

As already discussed, the publish/subscribe paradigm used in primary-copy schemes for update dissemination is typically implemented on top of a multicast mechanism. Thus, the performance of the multicast technique can greatly affect the cost and performance of the game environment. In this section, we look at various mechanisms and optimizations in more detail.

5.1. Direct Communication

In its simplest implementation, the primary copy sends all object updates to the nodes that have subscribed to the associated object. The main advantage of this approach is *low latency*. Updates on a local primary copy can reach the other clients in one virtual hop, therefore providing a better latency than client-server architectures where updates first go to the server for execution and then to the players. The other advantage is relative *simplicity*. However, this can result in high bandwidth requirements.

In fact, the *upload capacity* of peers can become a bottleneck as most broadband connections (e.g., DSL) are asymmetric, favoring download over upload capacity [Miller and Crowcroft 2010; Sundaresan et al. 2011; Otto et al. 2011]. The upload requirements can be calculated as the product of the number of clients that should receive the updates (subscribers), the update frequency, the object size, and the number of objects. These parameters greatly depend on the game. For example, in *Quake II* with an average object size of 200 Bytes, 10 updates per second, and between 8 to 64 objects, the upload requirements can be between 1 to 66 Mbps for 8 to 64 players [Bharambe et al. 2006](not using delta coding as explained later). The update frequency depends on the type of game. FPS games typically have a higher update frequency than RTS or RPG games. AOI filtering can reduce the number of clients receiving updates, thus reducing the upload requirements.

A particular problem is *interest heterogeneity* where a player becomes interesting to many other players, such as a player in possession of a valuable game item. These players have to send each update to many subscribers. An example is a “Capture the Flag” game where each team of players has to bring the other team’s flag to their own flag. There, flag carriers are focus points for all players, and the aggregate outbound traffic might surpass the carrier’s upload capacity [Bharambe et al. 2008].

Most unstructured architectures, as introduced in Section 4.3, rely on unicasts to all their neighboring nodes, for example, Hu and Liao [2004], Kawahara et al. [2004], Keller and Simon [2003], Bharambe et al. [2008], and Schmieg et al. [2008]. If the number of players in the AOI gets too large, the bandwidth requirements may exceed the available bandwidth. A recent study [Miller and Crowcroft 2010] using *World of Warcraft* logs showed that based on current residential broadband bandwidths available to users in the UK, direct P2P communication would result in saturation and high average latency.

In addition to AOI, there exist several basic mechanisms to reduce bandwidth requirements.

Delta Coding. It reduces the message size by only sending the differences (delta) in update messages. For instance, only the attributes of an object that have changed since the last update dissemination are sent. According to Bharambe et al. [2006], the size of an object update message can be reduced from 200 Bytes to 24 Bytes in *Quake II* using delta coding. As most games use UDP, update messages can be lost and delta coding can lead to incorrect state at secondary copies. Therefore, some games periodically send the complete object in order to correct the errors [Bharambe et al. 2008].

Exploiting Resource Heterogeneity. Simple forwarding mechanisms can be used when individual nodes become overloaded because of interest heterogeneity or because they have generally little upload capacity. In an extreme case, all message forwarding can be performed by a server or a communication hub, and only state computation is distributed across nodes, as described in Section 3.5. In a more distributed solution, more powerful peers can serve as forwarders for weaker peers [Banerjee et al. 2004; Bharambe et al. 2008; Schmieg et al. 2008]. pSense [Schmieg et al. 2008] uses an ad hoc approach. If a node does not have enough bandwidth to send an update to all neighboring nodes, it only sends it to a subset. In this case neighboring nodes help: whenever a node sends an update message it includes the list of nodes to which the message is sent. If a neighboring or sensor node receives a message and detects that it was not sent to a peer that might be interested, it will forward the message to this peer if it itself has enough bandwidth. This forwarding mechanism has its roots in probabilistic broadcast mechanisms [Eugster et al. 2003]. *Donnybrook* [Bharambe et al. 2008], on the other hand, deploys a forwarding pool consisting of nodes with good connectivity properties. Total bandwidth available to the forwarding pool should be able to support aggregate forwarding needs of other nodes. A machine becomes a part of the forwarding pool if its average latency to other nodes is below a certain threshold. In order to advertise their capacity, peers piggy-back their available bandwidth on a regular update message. A node that does not have enough bandwidth sends its update message to a random node in the forwarding pool which forwards it to other nodes. In each round a new random peer is chosen in order to eliminate the need for coordination and avoid overflowing a peer in the forwarding pool.

Message Aggregation. Multiple messages can be merged together to form a single message, thus reducing bandwidth requirements. Colyseus [Bharambe et al. 2006] aggregates overlapping subscriptions using a local subscription cache, which maintains subscriptions whose TTLs have not yet expired, and an aggregation filter, which takes multiple subscriptions and merges them if they contain sufficient overlap. Multicast trees, as will be explained in the next section, can specially benefit from message aggregation as all messages are sent to the root node of the multicast tree giving it a unique opportunity to merge them together. SimMud [Knutsson et al. 2004], for example, cuts the bandwidth requirements in half by using message aggregation. In many cases an artificial delay can be added before sending messages in order to allow for more messages to arrive and be aggregated together [Miller and Crowcroft 2010].

Selective Dissemination. The idea is to drop some events in overload situations instead of sending them, in order to maintain interactivity [Palazzi et al. 2006]. This idea is derived from the Random Early Detection (RED) [Floyd and Jacobson 1993] mechanism used by TCP to deal with congestion where overloaded routers randomly choose some packets to drop. In gaming environments, the focus is in avoiding to send events that have become obsolete. For example, a position update becomes obsolete once a new move action has been performed. Methods such as ILA-RED [Palazzi et al. 2006] and obsolescence detection [Ferretti 2008b] can be used to drop obsolete events. Gossiping mechanisms [Eugster et al. 2003] that forward messages from node to node can be especially prone to obsolete messages. Thus, messages should not be forwarded more than two to three hops. A specific form of selective dissemination is used by the 2003 version of *Solipsis* (denoted by Solipsis('03)) [Keller and Simon 2003], an unstructured P2P game, where in case of insufficient bandwidth a player reduces his AOI and drops some of the connections completely to better serve the players that remain in his reduced AOI.

5.2. Multicast Trees

Direct communication has its limitations. This is particularly true for coordinator-based architectures such as SimMud where superpeers have more responsibilities: they are in charge of a region and all communication for that region passes through these peers. In such situations, exploiting a P2P overlay for multicast purposes is attractive. A first possibility is to exploit IP-level multicast. For example, MiMaze [Lety et al. 1998; Diot and Gautier 1999], a 3D serverless Pacman game, uses UDP/IP multicast, the Real-Time Protocol (RTP), and the MBone to disseminate updates to all players. Similarly, DIVE [Frécon and Stenius 1998] relies on IP multicast and RTP. However, IP multicast requires support from routers which is not currently ubiquitous [Chu et al. 2002]. As a result, End System Multicast (ESM) schemes, implemented at the overlay level, are attractive [Chockler et al. 2001; Castro et al. 2002b; Chu et al. 2002; Macedonia et al. 1995].

Most common are logical multicast trees where tree nodes are peers in the system. A multicast message is sent to the root of the tree and from there sent by every peer to its children in the tree. The main advantage is *lower bandwidth requirements*: as each peer in the tree typically has few children it will have sufficient capacity for forwarding all messages, therefore alleviating the bandwidth problems. For example, Score [Lety et al. 2004] is a multicast-based communication protocol for large-scale virtual environments. To handle a large number of participants, it supports multiple multicast groups and multiple agents.

One drawback of this approach is *higher latency* as the number of virtual hops the message has to travel in order to reach the destination node is increased. This number is determined by the length of the path from the root to the destination node. Note that a virtual hop is a hop from one peer in the overlay to another one and may itself be comprised of several actual network hops. Therefore, messages can incur high latencies, and peers that are closer to the root will receive messages earlier than peers that are further away. Many multicast mechanisms try to maintain geographical locality in order to improve latency. As mentioned, games in particular are sensitive to delay, and high latencies render the architecture unusable.

A second problem is that many of the general-purpose multicast mechanisms build trees whose inner nodes might simply be forwarders not interested in the messages themselves, that is, they themselves are not subscribers to the updates. In addition to the increased network overhead (i.e., nodes sending and receiving messages they normally do not need to (in other architectures) for rendering the game), games in particular can suffer the possibility of unauthorized access, delaying the forwarding, or other malicious or selfish behaviors by forwarders (e.g., reading or dropping other players' messages; see Section 9).

Finally, maintaining multicast trees has computational and network overheads. While these issues hold in general for end-system multicast mechanisms, they are particularly problematic for games as they are very dynamic. For instance, in SimMud, whenever a player changes his region, he has to unsubscribe from one group and subscribe to a different group, leading to changes in the multicast trees. Similarly, when peers join or leave the game, adjustments have to be done. These changes can be very rapid depending on the speed of the game, and frequent group changes can undermine normal multicast operations. Similarly, since unsuitable multicast group sizes can have adverse effects on the performance, the games might have to adjust region sizes to make groups smaller, therefore providing a worse game experience.

Several methods have been proposed to address these drawbacks. The N-tree mechanism [GauthierDickey et al. 2005] divides the game world into nested regions when a region becomes overloaded. These nested regions are represented by a tree as shown in Figure 5(b). In addition, events are *scoped*. Scoped events are tuples consisting of actions, their location, and the scope of the event's impact. An event is sent along the tree to other peers in the region or regions according to its scope. Therefore, only nodes that are interested in the update will receive the update message. In Yamamoto et al. [2005], the world is split into regions similar to SimMud; however, a DHT is not used. A distributed event delivery system is proposed that alleviates the load of region servers by building multicast trees with intermediate nodes as it becomes necessary. MM-VISA [Ahmed and Shirmohammadi 2010], as discussed before, addresses rapid movements in a zone by clustering players together according to their characteristics.

Many ESMs, such as Scribe [Castro et al. 2002b], have built-in optimization in order to improve scalability and decrease the delay. Scribe employs measures to remove bottlenecks and collapse the long path by removing nodes that are not members of groups.

5.3. NAT and Firewalls

Clients are typically connected to LANs that are behind NATs (Network Address Translation) and firewalls. This can make inbound communications difficult. As a result, communication protocols designed for games have to accommodate these limitations. Since most games use UDP for update dissemination, protocols such as hole punching and STUN [Rosenberg et al. 2003] can be used for establishing connection mappings between peers [Seah et al. 2009]. Players unable to establish a connection between each other may rely on forwarding by other nodes or a server for communication. A study has shown that about 82% of the NATs tested support hole punching for UDP, and about 64% support hole punching for TCP streams [Ford et al. 2005].

6. INTEREST MANAGEMENT

As mentioned, interest management is tightly coupled with the zoning and architecture used. In keeping with the discussion so far we categorize the interest management techniques according to the architectures.

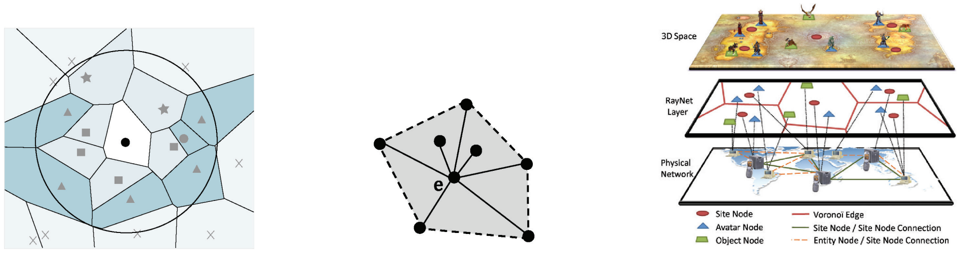
6.1. Structured Architectures

In many structured P2P games [Yamamoto et al. 2005; Knutsson et al. 2004; Iimura et al. 2004], the game world is divided into regions that are typically determined at the start of the game, and thus are static. The AOI is usually defined as the entire region, which allows the coordinator to simply multicast the message to all subscribers of the region. However, players may receive updates for objects that are not really of interest for them as they cannot be seen by the avatar. Alternatively, the AOI could be smaller than the entire region [Kulkarni et al. 2010]. In this case the coordinator would need to do message filtering and a simple multicast cannot be used anymore since dedicated messages have to be sent to each player. In any case, supporting visibility and interaction between players close to borders in different regions is either very hard or expensive. Moreover, a hand-off mechanism is necessary when a peer moves to a new region and changing the coordinator of a region can be complex. While these problems sometimes happen in unstructured networks as well, they are more common in structured systems.

Few structured gaming architectures are flexible in their interest management. One way to make region-based interest management more dynamic is to dynamically change the size of the region in question. N-trees [GauthierDickey et al. 2005] and P2P *Second Life* [Varvello et al. 2009a] dynamically divide a region into smaller regions as the region becomes overloaded, that is, the number of objects in the region is higher than a threshold. The drawback of such an approach is the increased cost of hand-offs to neighboring regions as regions become smaller. Another approach proposed by Badumna [Kulkarni et al. 2010] is hierarchical interest management: The game world is divided into cells, however, each player has a dynamic AOI, for example, in the shape of a sphere, inside the cell. This allows for more control and lower interest management overheads. In addition, players close to each other use a gossip protocol between each other to avoid sending too many queries to the coordinator and overloading it. In Colyseus [Bharambe et al. 2006] players use multi-attribute range queries to register their AOI with Mercury [Bharambe et al. 2004]. Whenever the AOI of a player changes, she subscribes to the new AOI by submitting a subscription request for the range query corresponding to the new AOI. She also unsubscribes from the old AOI. At subscription time, the player receives replicas of all objects that are in the new AOI if she doesn't have them yet.

6.2. Unstructured Architectures

Most unstructured gaming architectures provide dynamic AOI management. The principal idea of most approaches is that each player keeps a neighbor list of players and objects in his AOI and whenever players move, they send position updates to all neighbors. At the same time, a move changes the AOI of the player to some degree. Therefore, he has to get to know the players in the new area that was not covered by the old AOI.



(a) VON Voronoi diagram (b) Solipsis('03) convex hull (c) Solipsis('08) Raynet overlay

Fig. 6. (a) Neighbor discovery and Voronoi tessellation in VON [Hu et al. 2006] (©IEEE 2006). Different neighbors help in the process of neighbor discovery; (b) convex hull calculation by Solipsis('03) [Keller and Simon 2003] (with permission of Keller and Simon); (c) Solipsis('08) Raynet (Voronoi-based) overlay mapping [Frey et al. 2008] (with permission of Frey et al.).

The big difference between these mechanisms is in the shape of the underlying AOI, from a simple circle to more complex shapes, and how these shapes help in finding new neighbors.

Neighbor Detection. We further divide the neighbor discovery methods to two groups.

Tessellation techniques. Many approaches rely on a tessellation technique to determine the area belonging to a player and his neighborhood. For example, in ASCEND [Hu and Liao 2004], later VON [Hu et al. 2006], and VSM [Hu et al. 2008], each player computes a Voronoi diagram [Blau et al. 1992] based on his and other neighboring players' positions as shown in Figure 6(a). In the Voronoi region/cell, all points in the region are closer to the local node than any other node. In these networks nodes in a neighboring cell are connected to each other. When players move, they send position updates to all neighboring nodes causing them to recalculate their Voronoi diagrams. If a neighbor detects that a player's AOI is now overlapping with another neighbor, he sends a notification alerting the player about the new approaching player.

In Solipsis('03) [Keller and Simon 2003], each node calculates a convex hull of adjacent entities as shown in Figure 6(b). Nodes frequently query their neighboring nodes to inquire if they have detected any new entities in their AOI. A newer version of *Solipsis* in 2008 (denoted by Solipsis('08)) [Frey et al. 2008] relies on Raynet [Beaumont et al. 2007], an n-dimensional Voronoi-based overlay, to provide support for multi-participant virtual worlds (Figure 6(c)). It uses data dissemination policies that take advantage of the view-dependent representation of the 3D content. In addition, it distributes the execution of computationally intensive tasks. Varvello et al. [2009b] use a Delaunay network which is an overlay network whose topology is defined through Delaunay triangulation. The coordinates of avatars in the virtual world are used to generate the Delaunay triangulation and consequently connect neighboring nodes among the respective end-users.

Free format. These methods only rely on exchanging positions between players for neighbor detection. pSense [Schmieg et al. 2008] simply uses a circle with a given radius around a player as the player's AOI. In order to detect new neighbors, neighboring peers help out. If they receive a position update and realize that the sender should see a new player she is not yet aware of, they forward the message to the new player which can then initiate the interaction with the sender. In Kawahara et al. [2004], each peer keeps track of a fixed number of nearest nodes. Nodes constantly exchange their neighbor lists via gossiping and calculate and sort the distances (Euclidean distance in the game world) between entities. *N* nearest nodes are chosen as the active entities and the rest as latent entities that have potential for interaction.

6.3. Challenges

Structured and unstructured architectures each have several advantages and disadvantages.

Join process for new nodes. In structured networks, a join process is typically more streamlined (e.g., $O(\log n)$) but this comes at the higher costs of maintaining different routing tables. Whenever a new node joins the overlay, in addition to filling its routing tables, neighboring (in the overlay) nodes have to be notified and update their tables as well.

A problem with most of the unstructured approaches is that when a new node joins the system, it might take a long time to get to know its real neighbors if it only knows peers that are far away from its AOI. Same is true if a player teleports, that is, instantly moves to a far region.

Connectivity. In structured systems, the global managing system ensures accessibility and connectivity of the peers. However, the node location in the structured overlay may not directly correspond to the player's location in the game world and his neighboring players. This means the player will have to maintain many connections that are unnecessary from a game rendering point of view.

In unstructured systems, the overlay is more likely to become disconnected. As typically a finite number of nearest neighbors are maintained by a node, groups of users may lose contact with each other if they are separated by a large distance [Kawahara et al. 2004]. In particular, if there are several hotspots in the game world, different communities might become disconnected. pSense reduces the risk of isolation by requiring each node to connect to a sensor node in each section of area around it even if the sensor node may be far out of its AOI. The use of Voronoi diagrams or convex hulls effectively avoids disconnections but requires complex and process-intensive calculations. Moreover, the architectures are presented in 2D and are not easily extended to 3D without loss of scalability as these calculations do not scale well in 3D.

Recall. Due to their design, structured overlays provide better recall than most unstructured overlays, that is, if an object exists in the system, a request will most likely find it.

Overhead. Structured overlays rely on frequent messages, for example, heartbeat messages, to maintain connectivity and have performance optimization techniques to choose a closer neighbor (according to a performance metric) that itself relies on occasional measurement of networking performance of the neighboring nodes. A drawback of unstructured overlays is the overhead of frequent and explicit exchange of neighborhood lists between peers when gossiping is used [Kawahara et al. 2004; Keller and Simon 2003]. pSense includes the addresses of all neighboring nodes that are recipients of the update message in the message, resulting in a high overhead for each update message.

Multiresolution Interest Management. In all-to-all P2P game architectures other techniques than AOI have to be used for interest management. For instance, in open world maps or when regions are very big, players have a larger visibility range, and thus can see a larger number of players. Furthermore, if an avatar is in a popular area many other players will reside in his AOI even if it is small. In all these cases the traffic of this player might exceed his bandwidth capacity. To address this issue multiresolution techniques can be used that rely on sending updates at different rates and with variable details based on how interesting a player is.

MASSIVE [Greenhalgh and Benford 2002] differentiates between *focus*, the observer's allocation of attention, and *nimbus*, the observed object's manifestation or observability. The observer's awareness is a combination of these two factors.

Donnybrook [Bharambe et al. 2008] proposes *interest sets*. The reasoning is that human beings are only able to focus on a very limited number of items at once. Therefore, a player is only interested in the exact actions of few other players and will only notice basic information about other players in the visibility range. This fact can be exploited by creating high-fidelity replicas only for the players that the player has currently high interest in, while maintaining low-fidelity replicas of other players, which *Donnybrook* calls doppelganger. High-fidelity replicas are maintained by sending frequent updates while low-fidelity replicas receive *guidance* messages at a slow rate. In order to make doppelgangers look smooth, AI techniques are used to extrapolate realistic behavior between the reception of guidance messages (see Section 7). By keeping a fixed number of players in the interest set, the number of high-fidelity replicas does not depend on the size of the AOI or the number of players in the AOI but can be chosen to easily handle the bandwidth requirements. The frequency in which low-fidelity replicas are updated can be adjusted depending on the number of replicas in the AOI. Similarly, the use of different Level of Details (LoD) [Chien et al. 2010] has been proposed in the context of 3D content streaming as well as games hosted in the cloud [Najaran and Krasic 2010].

A question is how the game engine can determine which objects actually get the player's attention and need to be high-fidelity replicas. In order to estimate attention, *Donnybrook* uses the weighted sum of three parameters. *Proximity* reflects the fact that players tend to pay more attention to other players in their very close vicinity. *Aim* refers to first-person-shooter games, where players closely observe objects they are aiming at. It is determined by the angle between the player's view and the other players. Finally, *interaction recency* takes temporal locality into account as players who interacted recently are more likely to pay attention to each other.

7. REPLICATION AND CONSISTENCY CONTROL IN P2P ARCHITECTURES

Here we present a brief discussion on where the primary copies of players, NPCs, and other items are stored and how their replicas are managed.

7.1. Replication Management

As mentioned, most multiplayer games use a primary-copy scheme. Different architectures propose different locations for keeping the primary copy and how to send updates to replicas. Also, they provide different methods for keeping the game state as a whole consistent.

Location of Primary Copies. In systems where superpeers are coordinators for regions [Knutsson et al. 2004; Yamamoto et al. 2005; Yu and Vuong 2005] the primary copies of all objects in the region reside on the coordinator. This includes players, nonplayer characters, and other types of objects. The coordinator receives all update requests for these objects, executes them, and disseminates updates typically via multicast. In these systems, load balancing can be achieved by making regions smaller for the overloaded nodes [GauthierDickey et al. 2005], thus reducing the number of objects maintained by the coordinator. A particular problem can be NPCs, as they have to execute a think function to determine their next actions. Given that AI calculations can be processing intensive, this may lead to overloading the coordinator. The coordinator may decide to delegate this task to other players in the region [Iimura et al. 2004].

On the other hand, typically in mutual notification and all-to-all systems, each player is the holder of her avatar's primary copy and can perform updates locally. The update dissemination is done directly to other peers. Many of the unstructured systems, such as pSense or distributed avatar management [Varvello et al. 2009b], ignore the problem of state management for objects and NPCs. A possible solution, for example, in Voronoi-based architectures like Solipsis('08), is a state management scheme that puts primary copies of game objects and NPCs inside the (Voronoi) region onto the peer responsible for that region. In addition, Solipsis('08) distributes the heavy physics computations to allow for better scalability. One problem with such an approach is that primary copies of objects migrate too often between players as the player moves in the game world and her Voronoi regions change. Furthermore, NPCs can also change their positions frequently, leading to additional migrations. However, the fact that players in the neighboring regions already would have a replica of the objects and NPCs can help lower the object migration costs. VSM [Hu et al. 2008] provides a state management mechanism for Voronoi-based mutual notification systems. However, it relies on superpeers, called arbitrators or aggregators, for this task. Aggregators are responsible for multiple cells. An underload and an overload parameter are defined and the aggregator merges the load of underloaded peers or splits overloaded cells. The radius of an aggregator is determined according to its resources and shrinks as it becomes overloaded.

Fast Replica Management. In general, a player only maintains replicas of objects in his current AOI. When the AOI changes, the peer might have to receive new replicas. This can be time consuming and lead to lower game quality. To circumvent this problem, objects can be *prefetched*. Often, the direction a player is moving to can be predicted [Yahyavi et al. 2011]. Thus, replicas of relevant objects can be transferred before they actually enter the AOI [Bharambe et al. 2006]. One can also simply have an *area of subscription* that is larger than the AOI. Whenever an object is in this area of subscription, the player subscribes to it and receives a replica. Similarly, object replicas are only deleted when they are no more in this AOI for some time. This prevents thrashing behavior where a player continuously creates and deletes replicas due to small movements.

Standard object discovery might not be fast enough for short-lived objects in the game such as missiles in *Quake II*. These objects are created during the game and live only for short times. Therefore, prefetching does not work. Colyseus solves this through *proactive replication*. It attaches each short-lived object, for example, a rocket, to the long-lived object, for example, the player, that creates it. Players that are interested in the creator will automatically receive replicas of attachments when they are created. The same mechanism can be used for NPCs that are attached to player avatars such as companions or pets that accompany the player's avatar in the game. Since these companions are geographically close to the player their primary copies are typically assigned to the same node as the main avatar.

7.2. Consistency Control

As discussed in Section 2.8, various levels of consistency can be defined. Games can often tolerate relaxed forms of consistency control. The primary-copy model provides strong consistency control over an object (meaning concurrent and conflicting updates are not sent to replicas), however, replicas might have stale information.

Hiding Staleness. In order to hide the staleness of data, several mechanisms can be used. In Section 2.8 we mentioned dead reckoning and multiresolution mechanisms. These can also be applied in P2P architectures. Dead reckoning can be used for players that do not send their updates frequently. However, even though many approaches have been proposed to increase the accuracy, for example, AntReckoning [Yahyavi et al. 2012], dead reckoning is generally only able to tolerate delays of a few hundred milliseconds and is restricted to player movements. *Donnybrook*, on the other hand, uses a special replica called doppelganger. A doppelganger is an AI-controlled program aiming to be a realistic approximation of the remote player. For that the remote player sends infrequent guidance messages which include statistics about the player and the possible future moves. The AI tries to never deviate substantially from the guidance messages, and thus the character appears natural to players that are not paying close attention.

Multilevel Consistency Control. A particular challenge arises if an action updates more than one object as this requires coordination between different primary copies. Ideally such execution is transactional, providing atomicity and isolation. However, well-known protocols such as locking or quorum techniques are expensive, and most games accept weaker consistency levels.

In Zhang and Kemme [2011] the authors analyze various types of game actions and categorize them according to their consistency requirements. Different interactions have different levels of sensitivity and can be categorized accordingly. For each category, a consistency algorithm is provided. The two lowest levels are for actions on a single object. In the *no-consistency level*, replicas are only updated through a best-effort approach. Graphical effects could be actions for which this level is sufficient. The *low-consistency* level provides a bound on the staleness of the data. Progressive movements of players can fall in this category. Dead reckoning, for example, can be considered a low-consistency mechanism. The three next levels are for multi-object actions. They differ in the way in which they provide the transactional property *isolation*. *Medium-consistency* level does not provide isolation, in particular it allows stale reads, thus, actions might lead to results that are not anticipated by the player (e.g., picking up an empty bottle that the player thought to be full). *High-consistency* level provides isolation only in regards to some specifically marked attributes such as the price of an object. Finally, *exact-consistency* level provides complete isolation avoiding any form of stale reads. Clearly, the higher the consistency level is, the more costly the mechanism that achieves it.

Not all games use a classic primary-copy approach. Some allow updates to be submitted to any copy (e.g., for better responsiveness for the client). In this case, an inconsistency handling protocol is needed.

Optimistic Consistency Control. In many games, updates are performed optimistically on replicas at the time the update is submitted locally. This helps to hide the latency experienced by sending the updates to the primary. If the action leads to different changes at the replica and the primary, an inconsistency resolution protocol is necessary. One simple resolution mechanism, used for example in mobile games [Chandler and Finney 2005], could be to choose one of the states as the authoritative state, discarding all other inconsistent states. In Colyseus if there are concurrent updates, the last received update wins without any conflict resolution. An alternative form of conflict resolution is to roll back to older, consistent states. *Timewarp* [Jefferson 1985] is a rollback-based mechanism proposed for distributed simulation and databases, where all copies are allowed to execute updates optimistically. They keep track of old states, and roll back when inconsistencies occur. Such inconsistencies can occur, for example, if nodes receive out-of-order updates. *Timewarp* also periodically calculates Global Virtual Time (GVT) which is a time point in the system where it is guaranteed that no participant will roll back to a time before it. States with a time point before GVT can be removed as they will never be used. As rollbacks can be dissatisfying to players, several optimizations have been proposed over *Timewarp* to improve its performance in a game environment [Ferretti 2008b]. *Hydra* [Chan et al. 2007] proposes a new programming model to provide consistency guarantees when nodes fail in a P2P gaming architecture. This is achieved through a system of checkpointing and restoration of application game state as well as imposing conditions on message processing and message delivery.

8. FAULT TOLERANCE AND PERSISTENCE

8.1. Fault Tolerance

Generally, peers in a P2P system are much more prone to failures or unscheduled disconnections than servers in corporate data centers. This can result in the loss of the game state maintained by the failed peers or messages that are forwarded within a damaged multicast tree. In addition, it may result in loss of connectivity and isolation of peers.

Note that some of the general failure assumptions for P2P systems can be weakened in case of games, as discussed in Knutsson et al. [2004]: (1) As players in MMOGs are typically geographically diverse, node failures will be independent and it is unlikely that a large number of nodes fail simultaneously. (2) In short-lived games players typically stay until the end of the gaming sessions and do not suddenly leave. In continuously running games ungraceful exit from the game might be penalized as it can be considered as *escaping*. Thus, players are likely to leave in a controlled way allowing for a correct overlay and multicast tree reconfiguration. (3) Games usually provide only weak-consistency guarantees even if no node failures occur. For instance, sporadic message loss can be handled, inconsistent replicas are possible, and reconciliation mechanisms exist. Thus, failure handling becomes less critical.

In addition, many general-purpose DHTs and multicast trees have built-in mechanisms that repair the network overlay and the multicast trees in case of node failures. They typically rely on heartbeat messages sent frequently by nodes to each other in order to detect unresponsive (failed) nodes. MMOGs that are built over these existing P2P systems can take advantage of these mechanisms.

Primary Copy and Connection Loss. One main issue in case of node failures is the loss of the primary copies that resided on the failed node. In case of structured P2P games with region coordinators, a primary/backup mechanism can be used where the state at the coordinator is replicated on one or more backups that will take over in case of failure [Castro et al. 2002b]. As DHTs such as Pastry automatically send all messages that were directed to a failed node to the node with the closest NodeID, this node can be chosen as the backup of the coordinator, and failover occurs transparently. In Ahmed and Shirmohammadi [2010] and Ahmed et al. [2009] a hybrid approach is presented where player clusters are used to minimize the loss of connections and messages due to fast-moving players. In addition, two approaches are proposed to reduce the loss of connections. First, players are required to maintain redundant connections to their siblings and grandparents in the multicast tree. Second, a standby node is used to forward game states when needed.

In unstructured architectures peers often only hold the primary copies of their own game objects. Therefore, in case of failure only little game state is lost and the other peers can continue the game. In case nodes can hold primary copies of general game objects, for example, of objects in their Voronoi region, VSM [Hu et al. 2008] suggests to choose a random backup node for each node N in the overlay that is in charge of keeping replicas of all primary copies that node N holds. In case of failure, neighboring nodes will have to initiate primary copies for these general game objects and can receive the state by querying the corresponding replicas on the backup node.

Hybrid architectures might be able to rely on servers that are less likely to fail. Servers could maintain replicas of all objects, which can then be used to reinitialize failed primary copies.

Availability. Availability of the system is tightly coupled with fault tolerance. It depends on how well the failures of nodes are handled and the service is restored by other nodes taking over. Many P2P architectures have built-in mechanisms to determine availability of nodes, such as heartbeat messages. In addition, several availability monitoring and management systems have been proposed for both structured and unstructured P2P systems (not specific to games). *Total Recall* [Bhagwan et al. 2004] manages availability by adapting the degree of redundancy and frequency of repairs to the distribution of failures and can guarantee user-specified levels of availability. It performs three main tasks: (1) *availability prediction:* the system monitors the current state of the system and predicts future availabilities; (2) *redundancy management:* based on predictions, redundancy requirements are calculated; (3) *dynamic repair:* current availability monitoring information along with the predictions are used for dynamically repairing the system. *Total Recall* is built on top of a DHT substrate making it more compatible for use with the structured P2P architectures we have studied. Similarly, *AVMON* [Morales and Gupta 2009] provides a protocol for availability monitoring of the overlay in a scalable and efficient manner, that works with any arbitrary monitor selection scheme that is consistent and verifiable.

8.2. Persistence

Apart from replication, making game state persistent is another way to survive node failures. Even if replication can be leveraged for fault tolerance and quick fail-over, persistence remains an important building block to allow for game maintenance, to be able to recover from severe failures, and for very critical information for which maintenance within a transactional database system is necessary. Generally, persistence in P2P-based MMOGs has not received a lot of attention [Gilmore and Engelbrecht 2011].

What to store. Depending on the type of the game, the information that needs to be persistently stored is different. In a game like *World of Warcraft*, as avatars progress, their appearance, level, items, and location are typically made persistent as they are essential to the game. For games that are more short-lived, it is often game summary information, such as reputation of players, and their wins and losses, that are kept in persistent storage as these are the information that should survive the termination of the game session. In general, statistics, such as how successful players have been in different actions, might be recorded on stable storage, but not every detail about the actions. Zhang [2010] categorizes actions into persistence levels, similar to the consistency levels discussed in Section 7. Certain action types might never need to be written to stable storage, while for others it might be sufficient to write them asynchronously, that is, some time after they have occurred, which allows for aggregation, batch writes, and lower response times. Finally, for the most important actions it will be necessary to make them transactional, forcing persistence before the action concludes.

Persistence mechanism. The persistence mechanism is also dependent on the architecture used. In its simplest form, hybrid architectures can rely on the server component to provide persistence. If the server component is in any regard responsible for game state, that is, peers are only responsible for dissemination, this is straightforward. However, when game state is maintained by peers, that is, they have the primary copies, the server would need to keep replicas of objects in order to receive the updates, for example, by subscribing to the multicast groups, and then store them to permanent storage. The way the server subscribes to these updates, such as frequency and level of detail, should reflect the persistence categories of the items and actions in question.

Similarly, in superpeer approaches where coordinators are responsible for a region, the coordinator can be in charge for the persistence of its region. However, in pure P2P systems, game engines need to resort to a distributed permanent storage system.

There exist many solutions [Bhagwan et al. 2004; Rowstron and Druschel 2001b; Nagel et al. 2006; Chun et al. 2006] in structured or unstructured systems that provide these functionalities. Most approaches achieve availability and durability through redundancy: by keeping several copies of each data file so that it is not lost due to churn. For instance, *Total Recall* [Bhagwan et al. 2004] constantly monitors nodes in order to decide on the required replication degree and to dynamically repair lost copies. Other distributed storage systems take advantage of the particularities of structured DHTs, for example, in order to decide where to store which files. Architectures like Hampel et al. [2006] can use PAST [Rowstron and Druschel 2001b] to provide a persistent storage. PAST uses Pastry as the underlying content location and routing mechanism. It assigns a hash code as the file IDs to each file in order to distribute files. Caching and load-balancing techniques are also used to ensure efficient use of resources and the availability of the system.

Unstructured P2P systems typically use locality to maintain replicas for game items, therefore they cannot use a typical distributed file management system. However, they can rely on typical unstructured replica and file distribution techniques to create a distributed storage system [Nagel et al. 2006]. Carbonite replication algorithm [Chun et al. 2006] provides efficient replica maintenance for a distributed storage system. This is achieved by a range of techniques such as separating durability guarantees from availability, given that availability requirements are generally more expensive.

Gilmore and Engelbrecht [2011] provide an overview of persistence mechanisms in P2P architectures and compare different approaches for their suitability for P2P-based MMOGs.

9. CHEATING

Cheating is generally one of the main concerns in the design of a game architecture and one of the reasons game publishers have avoided the use of P2P architectures. Cheating is used by players to gain an unfair advantage over other players and to speed up their progress in the game. This results in noncheating players (also called *legit* players) becoming frustrated and either becoming a cheater themselves or leaving the game [Goodman and Verbrugge 2008]. Even though game companies are reluctant to publish data on cheating and security breaches, there is compelling evidence, such as users being banned from playing[7,8], that security problems and cheating are a serious issue. Frequent patches by game publishers, containing security fixes to prevent cheating, and disciplinary actions taken against cheaters, show that cheating is a pervasive threat. For example, over 35% of players in *Diablo* admitted to cheating at least one time. In addition, many companies have experienced losses in customer base due to high rates of cheating, that is, legit players leave the game due to frustration caused by players who cheat [Goodman and Verbrugge 2008].

P2P architectures, as a result of their design, are more prone to attacks and cheating than centralized systems [Chen and Maheswaran 2004a; Mogaki et al. 2007]. For instance, if the P2P overlay is used for update dissemination, players can cheat by withholding or accessing update messages, giving them an unfair advantage. If the P2P architecture provides distributed execution of updates and allows players to hold primary copies of game objects, cheaters can manipulate their object repositories and perform game state changes that violate game rules.

In this section, we present a definition of what constitutes cheating, different categories of cheating, and why P2P architectures are more vulnerable to cheating. We do not aim to provide a list of every single form of cheating but we focus on architecturally

[7] Bungie resets 15,000 Halo Reach players: http://www.bungie.net/Forums/posts.aspx?postID=49997802.

[8] Blizzard bans 5,000 StarCraft 2 cheaters http://us.battle.net/sc2/en/blog/882508.

more important cheats. From there, we discuss various cheat detection and prevention techniques that encompass different categories of cheating and architectures. This section only aims to give an overview of this important topic. Given the importance and range of the subject a detailed study of the subject seems appropriate and can be a survey paper in its own right [Webb and Soh 2007; Hoglund and McGraw 2007].

9.1. Definition

In this article, we use the definition of Goodman [2008] for cheating: “Cheating occurs when a player causes updates to game state that defy game rules and result in unfair advantage.” Cheating is possible as games are big and complex software systems and tend to have bugs and errors that can be *exploited* by users to give them an unfair advantage (as opposed to the developers’ intents). A game can be considered *fair* if the game state perceived by every player in the game is consistent with their expectations [Baughman et al. 2007]. In addition, cheating mechanisms can be based on *collusion* where several players collaborate for cheating purposes [Goodman 2008]. This collaboration can take place inside or outside the game world. For example, in a game of bridge, players might inform others of their hand.

Note that defying game rules alone may not be used as a complete measure since rules may not be clear to the players, resulting in unintended behavior. Furthermore, there are *emergent behaviors* where players are attracted to unintended aspects of the game, but this is typically not considered cheating. For example, in a racing game players may prefer crashing into each other rather than racing. Another concern is *social misconduct* which occurs when players do not break any game code rules but their behavior is nonetheless considered inappropriate. Abuse of other clients and other anti-social behaviors such as verbal abuse are examples of social misconduct. A well-known type of social misconduct in gaming is *griefing*[9] where a player deliberately tries to ruin other players’ game experience, for example, by stalking. As unintended and inappropriate behaviors are often very game specific, we will not discuss them further.

9.2. Cheating Categories

Cheating techniques can be categorized along various dimensions such as the layer in which cheating occurs (e.g., Webb and Soh [2007] and Yan and Randell [2005]), or along the architectural nature of cheating (e.g., Huguenin et al. [2011a], Goodman [2008], Baughman et al. [2007], Kabus et al. [2005], and Webb and Soh [2007]). We chose the latter, given our interest in the impact of P2P architectures on the vulnerability of multiplayer games. While many of these cheating techniques can occur even in client-server systems, we argue that in most cases P2P systems are more vulnerable to them. The taxonomy is fairly similar to the papers listed before.

Interrupting Information Dissemination. By delaying, dropping, corrupting, or changing the rate of the updates or sending wrong or inaccurate information, a player can blind or confuse other players about his current state. As a result, the chances of his avatar being targeted are decreased or he gains an unfair advantage in his attacks. Given that in P2P architectures nodes are in charge of disseminating their own updates as well as those of others, they are more vulnerable to these types of cheating. Specifically, nodes that are part of a forwarding pool or multicast tree can easily interrupt information dissemination. Several cheat types fall under this category.

—*Escaping.* The player intentionally terminates his connection in order to escape imminent loss in the game. Given higher failure rates of peers and relative lack of peer performance statistics in comparison to client-server systems, detection of escapes

[9] http://www.wowwiki.com/Griefing.

may be difficult. Detection techniques in this case may have overlap with availability monitoring techniques discussed in Section 8 [Yan and Randell 2005].

—*Time cheating (look-ahead cheating).* The player deliberately delays the updates he sends to base his actions on those he receives from others. Another form of this cheating uses fake *timestamps* in the past, where a player, after receiving updates from others, sends his own update with an old timestamp evading detection. This is only possible in a P2P system. In a client-server system the server sends state updates to all players after executing all the updates from the previous frame [Baughman et al. 2007].

—*Network flooding.* A player overflows the game server in order to create lags and disrupt game play. These attacks fall under Denial-of-Service (DoS) and Distributed-Denial-of-Service (DDoS) attacks and are widely studied. Many firewalls provide methods for detection and dealing with DoS attacks. Given relative lack of security measures at peers in comparison to a server as well as limited resources available to peers, they are more vulnerable to network flooding.

—*Fast rate cheat.* The cheater mimics a rate of game event generation that is faster than the real one, giving the player an unfair advantage in planning activities. Similar to escaping and look-ahead cheat, it is harder for peers to prevent and detect this behavior [Ferretti 2008a].

—*Suppress-correct cheat.* The cheater purposefully drops a number of consecutive updates and then sends an incorrect update that provides him with some advantage before being considered a dead peer. The incorrect update is harder to verify and dispute due to lack of recent information about the player. Given global knowledge of a server, it is in a better position to run an accurate validity test on delayed updates compared to normal peers [Baughman et al. 2007].

—*Replay cheat.* A cheater resends signed and encrypted updates of a different player that she has previously received [Corman et al. 2006]. For example, replaying a reduction in health can eventually lead to the death of the other player. In P2P systems, given that packets are forwarded by different peers, keeping track of the messages may become hard. In addition, forwarders are in a unique position to easily replay messages they forward.

—*Blind opponent.* A cheater drops some updates to opponents, blinding them about the cheater's actions, while still receiving updates from opponents [Webb et al. 2007]. This is only applicable to P2P systems as servers are typically trusted to send correct state updates at correct intervals. Similarly, by dropping update and guidance packets a player can make replicas of other players being guided by the AI (as in doppelgangers), acting *dumb* which can possibly give the player an advantage [Huguenin et al. 2011a].

Illegal Game Actions. A player can circumvent the game physical laws (e.g., limited velocity) and unduly change his state (e.g., increase his health or ammunitions) by tampering with the game code. A cheater can also unduly increase his score: in distributed FPS games, the players' scores are generally updated by having the players claim they have killed some other player, which constitutes a trivial opportunity to cheat, without proper verifications. Players may also use third-party software to perform game tasks such as aiming.

—*Client-side code tampering.* The player modifies the client-side code to get an unfair advantage. This is considered *abuse of authority*, and P2P systems are known to be vulnerable to these tactics. An example can be *speed-hacks* where the player tampers with the code to increase his avatars' movement speed. A number of such techniques are discussed in Feng et al. [2008].

—*Aimbots.* The player uses an intelligent program to provide her with automatic weapon aiming. Aimbots are mostly used in FPS games. Both P2P and client-server are vulnerable to this attack, however, lack of global knowledge and statistics about players can make detection harder in P2P systems.

—*Spoofing.* The cheater sends messages pretending to be a different player. Given limited authentication capabilities of P2P systems, they are more vulnerable than client-server systems.

—*Consistency cheat.* The cheater sends different updates to different players. This cheat can be used by a player or a group of players, and is only applicable to P2P systems.

Unauthorized Information Access. A player can exploit information available but not supposed to be disclosed (e.g., position of players behind walls) to increase his chances to kill other players or to foresee danger, thus helping him evade. This can occur as players might receive state updates or guidance messages about players that are not currently in their vision range. For example, players receive excess information to be able to quickly access the information necessary about other players should they enter their vision range, as in *prefetching* (see Section 7), therefore, On-Demand Loading (ODL) [Li et al. 2004] techniques are rendered inefficient. In addition, players may manage subscriptions to their avatars and, therefore be aware of avatars chasing them or aiming at them even if they are behind their backs. While untampered code might not show the information on the screen, such information can often be easily extracted. Moreover, the expected future position contained in dead-reckoning messages can be exploited by means of aim aides to increase shooting accuracy. One has to be aware that cheaters have access to information their colluding partners hold.

—*Sniffing.* Many tools, such as ShowEQ [2011], exist that allow players to log and access all kinds of information sent by the game across the network. This is a serious problem, especially if the communication is not done over a secure (encrypted) connection. Given that P2P systems receive extra information for neighbor discovery, forwarding, etc., they are more prone to these types of attacks than client-server systems.

—*Maphack.* The player, by tampering with the client code and libraries, is able to see through walls and obstacles, gaining an advantage. Many P2P systems, for example, mutual notification systems, receive excess information about other nodes to perform neighbor discovery, and are more vulnerable.

—*Rate analysis.* Multiresolution techniques, where updates are sent to interested players more often than the ones that are not interested, are prone to cheating by using traffic analysis. By analyzing the update packets being sent and the frequency of the updates, the player can find other players that are paying attention to him and try to escape [Huguenin et al. 2011b].

In general, many of the scalability and consistency techniques that we describe in this article are susceptible to cheating. Dead reckoning, for example, may be abused by users in suppress-correct cheat. Sending an update, after a number of dropped updates, causes the dead reckoning to apply the new update directly and makes it harder to detect cheating, giving the cheater unfair advantage.

9.3. Cheating Prevention

Cheating in P2P architectures has received some research attention. However, typically techniques proposed address a limited number of cheat types that can occur in specific architectures and scenarios. It is partly due to the fact that different architectures are more prone to different types of cheating. Some techniques try to *prevent* cheating from happening, that is, *proactive* mechanisms, while others *detect* cheating, that is, *reactive* approaches, and punish cheaters.

Prevention techniques proactively reduce or eliminate the possibility of cheating by players. These techniques fall under cheat prevention and proactive approaches.

Cryptographic measures. Cryptographic measures, such as message encryption, signatures, and checksums, are effective in eliminating message sniffing and illegal message modifications [Mönch et al. 2006; Eisenbarth et al. 2007]. Encryption ensures that forwarders do not have access to private data and cannot modify the messages. In order to protect against replay cheats, messages have to be uniquely identifiable. For example, Pittman and GauthierDickey [2011] use cryptographically secure hashes of cards to secure Trading Card Games (TCGs) eliminating the need for a referee (later explained). Cryptography mechanisms as well as message digests have been used to address replay and consistency attacks as well as spoofing [Corman et al. 2006; Chan et al. 2008].

Commitment and agreement protocols. Commitment and agreement protocols are another mechanism to prevent cheating, addressing a range of attacks. For instance, time cheating is addressed by the lockstep protocol [Baughman et al. 2007]. It requires all players to first submit a hash code of their next actions and only after every player's hash code has been received, players send their actions to each other. By comparing the hash code with the action, players can make sure that other players have not changed their actions after receiving input from others. This approach is quite similar to a two-phase commit protocol. Lockstep incurs very high latencies since players have to wait and receive all other players' hash codes before committing their own actions. This delay might not be tolerable by some games. Asynchronous synchronization [Baughman et al. 2007] tries to alleviate this problem by only requiring players within the area of interest to wait for one another. New Event Ordering (NEO) [GauthierDickey et al. 2004] improves this further by limiting the maximum latency and using a quorum of players for deciding whether an update from a certain player has been received in time or not. This causes players who have slow connections to most players to not be able to play. The Secure Event Agreement (SEA) protocol [Corman et al. 2006] improves NEO to address replay and consistency attacks as well as spoofing using changes in the cryptography mechanisms. EASES [Chan et al. 2008] improves SEA by computing a message digest before signing the message.

Many other infrastructures, with roots in agreement protocols, have also been devised. Chambers et al. [2005] address maphacks. In every frame, players exchange information about their viewable area. If they detect that they are in each other's viewable range they send their updates in plain text, otherwise, they only send a hash of their update. That is, updates are only sent once a player has verified that he is in the other player's view.

Other techniques. Several other techniques have been proposed to address a range of attacks. Here we present a few that address common cheating concerns such as aimbots. Golle and Ducheneaut [2005] and Schluessler et al. [2007] propose using CAPTCHA texts to deal with aiming bots. CAPTCHA texts are an example of a Turing test. However, distinguishing aiming bots from competent players based on the quality of their game play remains an open challenge. Hack-Proof Synchronization [Fung 2006] reduces the opportunity of cheating by using speed-hack in dead reckoning. Instead of directly including and sending the position and angular velocity of the player, other information is included in the message that can be used for dead reckoning. Huguenin et al. [2011b] prevent rate analysis cheats by using a proxy architecture where proxies forward messages and manage subscriptions of the interest sets of players. Proxies also hide dead-reckoning information from players unless they are in the vision range. Proxies are frequently changed in a random but verifiable fashion in order to decrease the chances of collusion and other types of cheating. Another popular method to prevent cheating is by choosing a superpeer for an area in which the superpeer itself is not playing and therefore is not interested in cheating. However, this method does not address collusion. CPP/FPS [Chen and Maheswaran 2004b] uses a *pulser* as an active authorized unit to broadcast pulse messages with the frame rate to synchronize the game pace.

9.4. Cheating Detection

Detection techniques are reactive and rely on the fact that if cheaters know that they will get caught and be punished, they will have less incentive to cheat. Punishment typically constitutes being banned from playing for a certain duration or indefinitely.

Verification. A common technique is that all actions are *audited* and *verified* for security breaches. For instance, a node that receives update messages from other nodes is able to verify them and check whether they are feasible according to the game physics or not and in essence act as a *referee*. Similarly, comparing hash messages of future updates with the actual updates is a form of verification that can detect cheaters [Baughman et al. 2007; GauthierDickey et al. 2004; Corman et al. 2006]. Such techniques have been used in other areas of distributed computing as well. For instance, Accountable Virtual Machines (AVMs) [Haeberlen et al. 2010] execute binary software images in a virtualized copy of a computer system and can record nonrepudiable information that allows auditors to check whether the software has behaved as intended or not.

Referee selection. The question now arises who can serve as referee. In coordinator-based architectures, since a centralized authority for each region exists, that is, the superpeer, most security checks can be performed by this node (similar to a server), assuming that this node has a high trust level [Izaiku et al. 2006]. Some hybrid systems such as Liu and Lo [2008] and Cecin et al. [2004] use players as referees that record and compute state updates. Cheaters are then reported to a trusted server. Referees can be chosen randomly to reduce the chances of collusion. In Webb et al. [2009], the authors propose a secure referee selection scheme. Referee Anti-Cheat Scheme (RACS) [Webb et al. 2007] is another hybrid approach that uses a secure server to validate all client updates to detect cheating.

Verification of computation. In coordinator-based games, the peers taking on coordinator tasks might not be trustworthy. Mutual checking schemes [Kabus et al. 2005] can be used by superpeers to ensure coordinators, given the same updates, reach the same state. Nodes periodically compare their states to detect cheaters. It can also be used to detect collusion between superpeers and normal peers. In hybrid systems such as DaCAP and FreeMMG [Liu and Lo 2008; Cecin et al. 2004] players are used as referees that record and compute state updates. Upon detection of cheating, the cheater is reported to a trusted server. In DaCAP monitors are chosen randomly to reduce the chances of collusion. RACS uses a secure server to receive, simulate, and validate all client updates to detect cheating. Clients are allowed to communicate directly with each other, however, they send an update to the server as the referee. Players that are detected as cheaters are only allowed to communicate through the referee. In Webb et al. [2009], the authors propose mechanisms to secure referee selection schemes. Another hybrid approach is Invigilation Reputation Security (IRS) [Goodman and Verbrugge 2008] in which communication is handled by the server and update execution is managed by peers. In this scheme the server assigns a proxy peer to each peer. Proxy peers are chosen randomly and are regularly reassigned. This proxy peer is responsible for executing the peers' updates and returning the results. Message relaying between peer and its proxy is done by the server. The server runs a quick auditing test to see if the returned result is possible according to the game rules or not. Conflicting updates and a percentage of updates randomly chosen are then reexecuted by a selected monitor peer to verify the results and detect cheating. In most of these systems, additional overhead is needed as update messages have to be sent to the chosen referees, in addition to the original target group.

When to verify. Another important question is when the verifications are performed. One possibility is to do so at the end of the gaming session. In this case little overhead is incurred during the game. However, this approach does not work in real time and cannot detect and ban cheaters during the gaming session. Real-time verifications are more effective but add overhead and latency to the game.

Verification process. The verification process itself can be very game and task specific: verifying whether movement speeds are accurate and actions are allowed given the characteristics and locations of players depends on the game itself. Ferretti [2008a] proposes a statistical method for detecting fast rate cheating by measuring the latencies between packets. Laurens et al. [2007] rely on behavioral monitoring to detect cheaters without any knowledge of the vulnerabilities in the game. By collecting information and statistics about the player's behavior through the game engine, anomalous behavior can be detected. In addition, Chen et al. [2008] propose AI bot detection by using a manifold learning approach. Two general classification techniques, *kNN* and Support Vector Machines (SVM), are used to differentiate AI and human behavior.

Other Detection Mechanisms. Apart from verification through referees, other detection mechanisms are possible. For instance, AC/DC [Ferretti and Roccetti 2006] proposes a method to detect look-ahead cheating in which a player that is suspicious of another player delays sending his action for a certain amount of time to see whether the suspected player waits for his action or not. In addition, several tools exist to detect tampering with the game code or to detect cheating processes running in the memory. PunkBuster [2011] and Valve Anti-Cheat (VAC) [VAC 2011] are examples of such systems. Closed and monitored environments such as XBox Live [XBox 2011] and PlayStation Network [PSN 2011] manage code tampering with regular updates to the gaming consoles. However, these tools can only detect a limited set of hacks and cheating tools, and always need to be updated to address a new cheating tool. In addition, simple methods such as occasional random aiming when using aimbots can result in not detecting a cheater [Webb and Soh 2007]. Randomly Created Checksum Algorithms (RCCAs) [Mönch et al. 2006] are used as an enforcement algorithm, by making its results necessary to gain access to the game data. RCCAs are deterministic checksum algorithms that are randomly created. Moreover, *Mobile Guards* are used to ensure the integrity of the protection mechanism. Mobile Guards are downloaded codes that contain the protection mechanisms and are not statically embedded into the game client. Feng et al. [2008] propose use of hardware-based, stealth measurements such as Intel's Active Management Technology (AMT)[10] that has been used in rootkit detection in operating systems. In this approach, a tamper-resistant processor resident on a client and isolated from the system's primary processor is used to perform randomly timed measurements *underneath* the host's software stack to detect cheats. Since the measurements are hardware based, they are more difficult to circumvent than if the checks were done by a software-based system.

[10] Intel vPro Technology http://www.intel.com/content/www/us/en/architecture-and-technology/vpro/vpro-technology-general.html.

The design of many cheat detection techniques is heavily dependent on the underlying P2P architecture and none of the techniques studied can address all possible cheating types. Often, a combination of these techniques has to be used.

Finally, once cheaters are detected and banned, one has to assure that they cannot reenter the game under a new alias. This is a question of authentication and persistence. It can be best handled if users have to join a game by going through a central authentication service.

9.5. Reputation Systems and Penalization

While prevention techniques are self-sufficient to address cheating, detection techniques rely on punishments to deter cheaters. As mentioned, thousands of players' accounts have been banned by the game developers such as Blizzard after running cheat detection algorithms [Goodman 2008; Egenfeldt-Nielson et al. 2008]. Unless the detection technique is deterministic, a reputation mechanism is required to find cheaters over time.

Reputation Systems. Particularly in architectures where peers are in charge of the verification and cheat detection, some form of a *reputation system* is required. Given that a single case of cheat detection may not always be conclusive evidence that a player is cheating, that is, it could be a false positive, it cannot be a basis for penalization. For example, a player may truly experience network problems and only send an update before being considered dead, similar to a suppress-correct cheat. In such a case, one needs to keep a record of players over time to be able to differentiate cheaters from honest nodes facing connectivity issues. Only after the number of detections passes a threshold it can be said with high probability that a player is a cheater. This threshold can be calculated based on how likely it is for normal behavior to be mistaken as cheating.

Several general-purpose distributed reputation systems have been proposed. These systems are in essence similar to the eBay [2011] rating system where buyers and sellers are able to rate each other. The overall reputation of a participant is the overall sum of these ratings in the last six months. EigenTrust [Kamvar et al. 2003] is an example of a popular distributed mutual rating mechanism designed for file sharing systems. Similar gaming reputation systems also exist. PlayerRating [Kaiser and Feng 2009] and REPS [Huang et al. 2008a, 2008b] are distributed game rating management systems. These reputation systems use the same basic rules: after each interaction, participants in the interaction rate each other and store the rating. Queries sent to friends and certain groups of players as well as some random peers allow these ratings to be collected and aggregated for different players, giving them an overall reputation. Reputation systems themselves are vulnerable to several attacks such Sybil attacks [Douceur 2002] where the attackers, by creating a large number of pseudonymous entities, gain a disproportionately large influence in the ratings. Centralized systems, deal with this by requiring verifications through phone numbers, credit cards, or other forms of identification. This can possibly be used by trusted peers as well. Badmouthing and collusion between players can also subvert the reputation system, which has been studied [Swamynathan et al. 2008, 2010].

Other approaches such as proactive reputation [Swamynathan et al. 2008, 2010] provide reputation systems that do not rely on third-party ratings and provide measures to evaluate reputation of peers directly. In proactive reputation, transactions are anonymized and cannot be distinguished from normal requests. This prevents peers from trying to boost their reputation. In Wierzbicki and Kaszuba [2007] authors propose a trust management system without the need for a reputation system given trusted superpeers. In addition, commercial third-party player ranking solutions, such as GameRank [2011], have been proposed that maintain a score for players by analyzing their activities in the game. Hybrid architectures, such as Liu and Lo [2008], Cecin et al. [2004], and Quazal [2011], can use the server for authentications and the subsequent punishments of the players.

Penalization. Penalization typically happens in the form of expelling players from their current games and/or banning them from accessing future sessions. In superpeer-based architectures, the coordinator, after detection of cheating by itself or other nodes, can ban the player from the game by stopping update dissemination to this node and ignoring its updates. However, measures have to be taken to ensure that a coordinator does not abuse this power [Izaiku et al. 2006; Kabus et al. 2005]. In mutual notification systems neighboring nodes have to all participate in banning a player from the game. In addition, cheaters have to be blacklisted (e.g., by distributing and storing their identifying information in the persistent storage) by peers to prevent them from reentering the game. However, given the relative ease of changing aliases, this can be difficult. In hybrid systems, players may have to go through the central server for authentication and can then join games. As a result, cheaters can be easily kicked out of a game or be banned by the server for a certain duration or indefinitely. Given a server's better authentication and persistence capabilities, it's harder for cheaters to leave and join the game under a new alias.

10. COMMERCIAL ADOPTION

Generally, P2P approaches are not widely adopted in commercial systems. In this section, we discuss some existing engines that use P2P solutions or that we believe can be extended for P2P support. Then we discuss why industry has not yet widely adopted the P2P model and what would be possible options for a better market penetration. This is followed by a discussion of how users could be given incentives to contribute their resources to P2P systems. Finally, we discuss how the P2P solutions for MMOGs presented here could be applied to other applications.

10.1. Middlewares

In most cases, game functionalities such as replication management, interest management, and update dissemination are implemented in a game engine that presents a middleware [BigWorld 2011; Henning 2004; Denault et al. 2008] to the programmer. It hides the complexities of the underlying architecture while providing an adequate programming interface to game designers. Many different games can be implemented using the same middleware.

10.2. Industry Solutions

Many network engines, or SDKs that include a network engine, have been proposed for multiplayer gaming (mostly centralized) which provide easy-to-use APIs that have built-in support for many of the common functionalities required for gaming. These functionalities include: reliable UDP streams, delta coding, NAT hole punching, dead reckoning, object synchronization, and others. Examples of such network engines are: Zoidcom[11], ClanLib[12], NLNet[13], RakNet[14], OpenTNL[15], and OpenSkies[16]. As another

[11] zoidcom.com.
[12] clanlib.org.
[13] opennel.org.
[14] raknet.com.
[15] opentnl.org.
[16] openskies.net.

example, BigWorld[17] provides a middleware that supports shards as well as zoning of worlds utilizing a dynamic real-time load-balancing system to remove zone limitations. In addition, similar to the multiresolution schemes mentioned before (see Section 6), it provides *Level of Detail (LOD)* and data prioritization that is applied to all object property changes and events to make sure that only the most important data is sent to the client.

While most of these network engines are designed and optimized for the client-server model, many of the functionalities, such as object synchronization and reliable UDP streams, can be used in a P2P engine as well. NAT hole punching, in particular, is of utmost importance in P2P networks since it is essential to be able to directly connect players behind NATs to each other. General-purpose APIs, for example, to create sessions, can be used in P2P networks but they lack necessary optimizations to address P2P issues and do not benefit from P2P network strengths.

RakNet with additional plugins provides support to connect all peers automatically. ReplicaNet[18] allows for creating network sessions and session migrations for both client-server and P2P network topologies. Ludiloom[19] uses a hybrid P2P architecture for efficient game content distribution and high availability. OpenSkies implements an MMO networking and simulation architecture to improve the real-time massively multiplayer capabilities of a High-Level Architecture (HLA), the IEEE standard [IEEE 2000] framework that supports modeling and simulation. It provides both a P2P implementation for small-scale multiplayer networking as well as a peer-to-lobby manager/router network implementation (lobby is explained further shortly). Furthermore, it allows clients to communicate with each other via external servers (federation hosts). Quazal Net-Z and Rendez-Vous [Quazal 2011] provide an optimized routing system for P2P connectivity as well as a game lobby service to provide matchmaking services between players. Game companies can use the lobby to ensure players pay to gain access to games and to authenticate players. The lobby provides access to different game sessions for players. The game sessions themselves are managed by P2P communication between clients, minimizing costs for the game companies. Badumna Network Suite[20] provides the necessary APIs for a scalable service for game state synchronization and object replication by using a decentralized network according to the designs in Kulkarni et al. [2010].

10.3. Industry Models

As mentioned, industry adoption of P2P networks has been limited. More than the *technical difficulties* as discussed so far in this article, the challenge remains in the problem of maintaining *control* over the game and dealing with *cheating*. Therefore, hybrid architectures are attractive since they allow to maintain control and can exploit the benefits of P2P architectures. We discuss some of the more attractive options to companies.

—*Game lobby model.* In this model the game company provides a matchmaking service that allows players to meet and play together. By enforcing all players to go through the lobby, the game lobby can ensure that the players meet the conditions of the game company, have paid subscription fees, and are authenticated and authorized to access the game sessions. In addition, it provides a means for punishing cheaters and preventing them from gaining access to games. This model minimizes the costs

[17] bigworldtech.com.
[18] replicanet.com.
[19] ludiloom.com.
[20] scalify.com.

as providing a game lobby is not expensive and the games are mostly managed by the clients. Quazal provides a sample architecture for such a system. This model is currently mostly used for relatively short-lived games.

—*Federated servers model.* Many of the architectures discussed in this article use a hybrid approach with superpeers where some high-level tasks can be delegated to the federated servers, while other tasks are performed by peers. These servers may belong to the same company or not. The distributed design may also be exploited by publishers to benefit from third-party federated servers. For example, the Badumna network can provide such a solution. Similarly, Content Distribution Networks (CDN) such as Akamai and Limelight and hybrid cloud and P2P-based games can also benefit from these architectures [Carlini et al. 2010]. Similarly, edge servers [Gao et al. 2003] (where inter-server communication is managed in a P2P fashion) can be used by CDNs and live game streaming companies such as Gaikai and Onlive [Gaikai 2011; OnLive 2011] to maintain game state close to the geographically distributed clients to reduce the latency or to provide 3D streaming services [Chien et al. 2010; Hu et al. 2010; Wu et al. 2009]. Moreover, in-browser gaming that typically relies on HTML5, *web players*, is becoming more popular [BigWorld 2011] making it suitable for 3D streaming. Similarly, using these technologies, the game can be executed in the cloud and players can use their PC, console, or browser as a terminal connected to the cloud [Najaran and Krasic 2010].

—*Free model.* Players are in charge of maintaining the game state and playing is free of charge for players. It can be used by the open-source community to create free games. It can also be attractive to companies to provide low-cost *Free-to-Play* (F2P) games, relying on selling in-game items for profit that has proven to be a very successful model[21,22]. Given the low cost of maintenance it can reduce the risk of investment. In addition, existing P2P architectures that already support apps, such as Skype[23] and μTorrent[24], can use games to make their platforms more attractive. They can use their already existing and powerful P2P network to support games, however, they need to adapt the game architecture to the network.

10.4. Client Incentives

Here we present several reasons why the P2P architectures can be attractive to consumers and to incentivize them to contribute resources to the system. Clients have several incentives to use a P2P game environment.

—*Better game environments.* P2P architectures have the potential to create more engaging game environments by removing many of the limiting factors in the game design. For example, higher number of players can be allowed in a single region of the game, leading to more engaging gameplay.

—*Lower fees.* Reducing costs can allow game developers to reduce the subscription fees, therefore, reaching a wider costumer base. In addition, schemes that adjust the fees according to the resources players can contribute to the game are possible. This is in particular attractive for free-to-play MMOGs that are supported by advertising and purchases of in-game items.

[21] €1000 Virtual Item For Game Raises €2 Million In 4 Days For Bigpoint: http://techcrunch.com/2011/11/23/e1000-virtual-item-for-game-raises-e2-million-in-4-days-for-bigpoint/.

[22] 4 in 10 playing freemium games make in-game payment to extend or enhance the game https://www.npd.com/wps/portal/npd/us/news/pressreleases/pr_120423a.

[23] skype.com.

[24] utorrent.com.

In order to ensure that players contribute resources to the game *distributed accounting mechanisms* can be used. Many Tit-for-Tat (TfT) mechanisms, first proposed for file sharing systems [Cohen 2003], have been proposed for P2P systems that force nodes to contribute to the system in exchange for using the system [Li et al. 2008]. Free rider detection mechanisms, such as Guerraoui et al. [2010], can detect nodes that are mostly consumers and do not contribute to the system. Unlike file sharing systems, creating incentives and punishments are easier in games. Distributed accounting systems such as Gupta et al. [2003] exist that use a credit/debit system that tracks nodes' activities. Nodes are billed according to their use of resources, and are credited based on the time and the quality of resources they have provided. By implementing such systems, free riders can be punished through mechanisms such as limited access to some parts of the game world, limiting the gameplay duration, and limited access to game items. Contributors, or players who actually pay for credits, can receive better reputation, experience points, or gain access to exclusive items. However, using these methods, one must ensure the gameplay is not too unfair to players with limited resources available by finding a suitable balance based on the game.

10.5. Other Applications

In addition to games, other applications may also benefit from the P2P techniques presented in this article. Note that the discussions also apply to 3D virtual environments other than games[25]. We ignore other applications of P2P networks such as delivering game software updates.

—*Mobile Multiplayer Games.* The emergence of mobile platforms, such as smartphones, Sony PSP, and Nintendo DS that integrate various wireless network capabilities, enables distributed multiplayer gaming on these platforms. However, the limitations in connectivity, latency, memory, and processing power make it unlikely that mobile platforms will become powerful peers in a massively multiplayer game. In addition, special mechanisms are needed to handle higher faults and increase latency tolerance. In particular, inconsistencies due to message delays and loss should be hidden and eventual consistency must be achieved without requiring large computational costs at the mobile platform [Chandler and Finney 2005; Korhonen and Koivisto 2007; Wang et al. 2009; Harvey et al. 2010]. Peer-to-peer mechanisms can be attractive to avoid costly communication among the mobile unit and an external powerful server when players are physically close. Mobile devices can use IEEE 802.15 or similar low-power communication interfaces and create ad hoc networks to perform some of the communication locally between players in a P2P fashion.

—*Social Games.* A new emerging area of games is *social games*. These games are typically built on social network platforms. Many of them are multiplayer in nature and as millions of players simultaneously play these games, they can be considered a form of massively multiplayer online game. However, currently these games typically lack a graphically intensive and immersive gameplay and are different in nature than the games studied in this article. These games may be asynchronous (turn based) and typically rely on social interactions between players, such as exchange of gifts and virtual goods, as one of the main game functionalities. These games rely on the social platforms for their operations. Architectures have been proposed to enhance the social networks built around MMOGs. CAMEO [Iosup et al. 2010] provides an architecture for continuous analysis for MMOGs using cloud resources, such as, Amazon EC2, to increase the revenue from advertisements or selling virtual goods and services in the social network communities.

[25] For a list see: http://www.virtualworldsreview.com.

Second Life can be considered the largest immersive social network environment. Research has shown that avatars in *Second Life* tend to interact similarly to human beings in real life. They meet, spend time together, and make friends. This behavior suggests that avatars construct an online social network that represents the social relationships existing among human beings [Varvello and Voelker 2010]. As demonstrated in Varvello et al. [2009a] P2P systems can be used in such systems. In addition, in such systems closeness is not only measured as the physical distance in the virtual world but also considers how close two players are in the social graph, or how many interactions they had in the past. P2P architectures can exploit this closeness measure as a factor for task distribution and cheat detection and prevention. Integration of social games and traditional MMOGs [Varvello et al. 2009a] can further make the use of P2P architectures possible.

11. CONCLUSION

Massively multiplayer online games are a thriving business industry and they introduce a range of interesting challenges. Most MMOGs follow a client-server architecture but there has been increasing interest in designing P2P or hybrid architectures for games. P2P architectures reduce the cost of maintaining expensive servers and improve the scalability of games.

P2P architectures are used in state distribution as well as update dissemination to peers. These architectures can be divided into three categories: structured, unstructured, and hybrid. While P2P architectures can provide higher scalability and lower cost they have to address many challenges such as limited bandwidth, or vulnerability to cheating. Many techniques have been proposed to solve these challenges, but many topics such as availability, persistence, and cheating remain to be fully addressed. In addition, while some of these techniques can be applied to different architectures, many are dependent on the underlying architecture. In this article, we do not present all available architectures, however, we believe each type of architecture is studied with sufficient related work to provide a comprehensive study of different techniques using a uniform terminology.

ACKNOWLEDGMENTS

We appreciate the detailed and helpful feedback from the anonymous CSUR reviewers that has greatly helped in improving the final version of this article.

REFERENCES

ADVE, S. V. AND GHARACHORLOO, K. 1996. Shared memory consistency models: A tutorial. *IEEE Comput. 29*, 12, 66–76.

AHMED, D. AND SHIRMOHAMMADI, S. 2010. A fault tolerance procedure for p2p online games. In *Proceedings of International Conference on Information Sciences Signal Processing and their Applications (ISSPA'10).* 614–617.

AHMED, D. T., SHIRMOHAMMADI, S., AND OLIVEIRA, J. C. 2009. A hybrid p2p communications architecture for zonal mmogs. *Multimedia Tools Appl. 45*, 313–345.

ARMITAGE, G. 2003. An experimental estimation of latency sensitivity in multiplayer quake 3. In *Proceedings of the IEEE International Conference on Networks (ICON'03).* 137–141.

BANERJEE, S., LEE, S., BRAUD, R., BHATTACHARJEE, B., AND SRINIVASAN, A. 2004. Scalable resilient media streaming. In *Proceedings of the International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV'04).* ACM Press, New York, 4–9.

BAUGHMAN, N., LIBERATORE, M., AND LEVINE, B. 2007. Cheat-proof playout for centralized and peer-to-peer gaming. *IEEE Trans. Netw. 15*, 1–13.

BEAUMONT, O., KERMARREC, A. M., AND RIVIERE, E. 2007. Peer to peer multidimensional overlays: Approximating complex structures. In *Proceedings of the International Conference on Principles of Distributed Systems (OPODIS'07).* Springer, 315–328.

BENFORD, S. AND FAHLEN, L. 1993. A spatial model of interaction in large virtual environments. In *Proceedings of the International European Conference on Computer-Supported Cooperative Work (ECSCW'93).* Kluwer, 109–124.

BERGLUND, E. J. AND CHERITON, D. R. 1985. Amaze: A multiplayer computer game. *IEEE Softw. 2*, 3, 30–39.

BHAGWAN, R., TATI, K., CHENG, Y. C., SAVAGE, S., AND VOELKER, G. M. 2004. Total recall: System support for automated availability management. In *Proceedings of International Conference on Networked Systems Design and Implementation (NSDI'04).* USENIX, 25–25.

BHARAMBE, A., AGRAWAL, M., AND SESHAN, S. 2004. Mercury: Supporting scalable multi-attribute range queries. In *Proceedings of the ACM SIGCOMM Conference.* ACM Press, New York, 353–366.

BHARAMBE, A., DOUCEUR, J., LORCH, J., MOSCIBRODA, T., PANG, J., SESHAN, S., AND ZHUANG, X. 2008. Donnybrook: Enabling large-scale, high-speed, peer-to-peer games. In *Proceedings of the ACM SIGCOMM Conference*. ACM Press, New York.

BHARAMBE, A., PANG, J., AND SESHAN, S. 2006. Colyseus: A distributed architecture for online multiplayer games. In *Proceedings of the International Conference on Networked Systems Design and Implementation (NSDI'06).* USENIX.

BIGWORLD. 2011. BigWorld technology. http://indie.bigworldtech.com/.

BLAU, B., HUGHES, C. E., MOSHELL, M. J., AND LISLE, C. 1992. Networked virtual environments. In *Proceedings of the International Symposium on Interactive 3D Graphics.* ACM Press, New York, 157–160.

BLIZZARD. 2011. Blizzard entertainment. http://www.worldofwarcraft.com/pvp/battlegrounds.

BOULANGER, J. S., KIENZLE, J., AND VERBRUGGE, C. 2006. Comparing interest management algorithms for massively multiplayer games. In *Proceedings of the International ACM SIGCOMM Workshop on Network and System Support for Games (NETGAMES'06).* ACM Press, New York, 1–6.

BUTTERFLY.NET. 2003. The butterfly grid: A distributed platform for online games, later emergent game technologies. http://www.gamebryo.com/.

BUYUKKAYA, E. AND ABDALLAH, M. 2008. Efficient triangulation for p2p networked virtual environments. In *Proceedings of the International ACM SIGCOMM Workshop on Network and System Support for Games (NETGAMES'08).* ACM Press, New York, 34–39.

BUYUKKAYA, E., ABDALLAH, M., AND CAVAGNA, R. 2009. VoroGame: A hybrid p2p architecture for massively multiplayer games. In *Proceedings of the International IEEE Conference on Consumer Communications and Networking (CCNC'09).*

CARLINI, E., COPPOLA, M., AND RICCI, L. 2010. Integration of p2p and clouds to support massively multiuser virtual environments. In *Proceedings of the International ACM SIGCOMM Workshop on Network and System Support for Games (NETGAMES'10).* ACM Press, New York, 17:1–17:6.

CASTRO, M., DRUSCHEL, P., GANESH, A., ROWSTRON, A., AND WALLACH, D. S. 2002a. Secure routing for structured peer-to-peer overlay networks. *ACM SIGOPS Oper. Syst. Rev. 36,* SI, 299–314.

CASTRO, M., DRUSCHEL, P., KERMARREC, A. M., AND ROWSTRON, A. I. T. 2002b. SCRIBE: A large-scale and decentralized application-level multicast infrastructure. *IEEE J. Selected Areas Comm. 20,* 8, 1489–1499.

CECIN, F. R., REAL, R., DE OLIVEIRA JANNONE, R., RESIN GEYER, C. F., MARTINS, M. G., AND VICTORIA BARBOSA, J. L. 2004. FreeMMG: A scalable and cheat-resistant distribution model for internet games. In *Proceedings of the International Symposium on Distributed Simulation and Real-Time Applications (DS-RT'04).*

CHAMBERS, C., FENG, W.-C., FENG, W.-C., AND SAHA, D. 2005. Mitigating information exposure to cheaters in realtime strategy games. In *Proceedings of the International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV'05).* ACM Press, New York.

CHAN, L., YONG, J., BAI, J., LEONG, B., AND TAN, R. 2007. Hydra: A massively-multiplayer peer-to-peer architecture for the game developer. In *Proceedings of the International ACM SIGCOMM Workshop on Network and System Support for Games (NETGAMES'07).* ACM Press, New York, 37–42.

CHAN, M. C., HU, S. Y., AND JIANG, J. R. 2008. An efficient and secure event signature (eases) protocol for peer-to-peer massively multiplayer online games. *Comput. Netw. 52*, 1838–1845.

CHANDLER, A. AND FINNEY, J. 2005. On the effects of loose causal consistency in mobile multiplayer games. In *Proceedings of International ACM SIGCOMM Workshop on Network and System Support for Games (NETGAMES'05).* ACM Press, New York, 1–11.

CHEN, B. AND MAHESWARAN, M. 2004a. A cheat controlled protocol for centralized online multiplayer games. In *Proceedings of the International ACM SIGCOMM Workshop on Network and System Support for Games (NETGAMES'04).* ACM Press, New York.

CHEN, B. D. AND MAHESWARAN, M. 2004b. A fair synchronization protocol with cheat proofing for decentralized online multiplayer games. In *Proceedings of the International Symposium on Network Computing and Applications (NCA'04).* 372–375.

Chen, J., Wu, B., Delap, M., Knutsson, B., Lu, H., and Amza, C. 2005a. Locality aware dynamic load management for massively multiplayer games. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'05)*. ACM Press, New York, 289–300.

Chen, K., Pao, H. K., and Chang, H. C. 2008. Game bot identification based on manifold learning. In *Proceedings of the International ACM SIGCOMM Workshop on Network and System Support for Games (NETGAMES'08)*. ACM Press, New York.

Chen, K.-T., Huang, P., Huang, C.-Y., and Lei, C.-L. 2005b. Game traffic analysis: An mmorpg perspective. In *Proceedings of the International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV'05)*. ACM Press, New York, 19–24.

Chen, P. and El Zarki, M. 2011. Perceptual view inconsistency: An objective evaluation framework for online game quality of experience (qoe). In *Proceedings of the International ACM SIGCOMM Workshop on Network and System Support for Games (NETGAMES'11)*. ACM Press, New York.

Chien, C. H., Hu, S. Y., Jiang, J. R., and Cheng, C. W. 2010. Bandwidth-aware peer-to-peer 3d streaming. *Proc. Int. J. Adv. Media Comm. 4*, 4, 324–342.

Chockler, G. V., Keidar, I., and Vitenberg, R. 2001. Group communication specifications: A comprehensive study. *ACM Comput. Surv. 33*, 4, 427–469.

Chu, Y., Rao, S. G., Seshan, S., and Zhang, H. 2002. A case for end system multicast. *IEEE J. Selected Areas Comm. 20*, 8, 1456–1471.

Chun, B. G., Dabek, F., Haeberlen, A., Sit, E., Weatherspoon, H., Kaashoek, M. F., Kubiatowicz, J., and Morris, R. 2006. Efficient replica maintenance for distributed storage systems. In *Proceedings of the International Conference on Networked Systems Design and Implementation (NSDI'06)*. USENIX, 4–4.

Cohen, B. 2003. Incentives build robustness in bittorrent. In *Proceedings of the Workshop on Economics of Peer-to-Peer Systems*. Vol. 6, 68–72.

Corman, A., Douglas, S., Schachte, P., and Teague, V. 2006. A secure event agreement (sea) protocol for peer-to-peer games. In *Proceedings of the International Conference on Availability, Reliability and Security (ARES'06)*. 34–41.

Costa, P., Migliavacca, M., Picco, G. P., and Cugola, G. 2003. Introducing reliability in content-based publish subscribe through epidemic algorithms. In *Proceedings of the International Workshop on Distributed Event-Based Systems (DEBS'03)*. ACM Press, New York, 1–8.

Denault, A., Kienzle, J., Dionne, C., and Verbrugge, C. 2008. Object-oriented network middleware for massively multiplayer online games. Tech. rep., SOCS-TR-2008.5 McGill University, Montreal, Canada.

Diot, C. and Gautier, L. 1999. A distributed architecture for multiplayer interactive applications on the internet. *IEEE Netw. Mag. 13*, 4, 6–15.

Douceur, J. R. 2002. The sybil attack. In *Proceedings of the International Workshop on Peer-to-Peer Systems (IPTPS'02)*. USENIX, 251–260.

Ebay 2011. ebay. http://www.ebay.com/.

Egenfeldt-Nielson, S., Smith, J. H., and Tosca, S. P. 2008. *Understanding Video Games: The Essential Introduction*. Routledge.

Eisenbarth, T., Kumar, S., Paar, C., Poschmann, A., and Uhsadel, L. 2007. A survey of lightweight-cryptography implementations. *IEEE Des. Test 24*, 522–533.

Eugster, P. T., Guerraoui, R., Handurukande, S. B., Kouznetsov, P., and Kermarrec, A.-M. 2003. Lightweight probabilistic broadcast. *ACM Trans. Comput. Syst. 21*, 4, 341–374.

EVE. 2011. EVE online. http://www.eveonline.com.

Feng, W., Kaiser, E., and Schluessler, T. 2008. Stealth measurements for cheat detection in on-line games. In *Proceedings of the International ACM SIGCOMM Workshop on Network and System Support for Games (NETGAMES'08)*.

Ferretti, S. 2008a. Cheating detection through game time modeling: A better way to avoid time cheats in p2p mogs. *Multimedia Tools Appl. 37*, 3, 339–363.

Ferretti, S. 2008b. A synchronization protocol for supporting peer-to-peer multiplayer online games in overlay networks. In *Proceedings of the International Workshop on Distributed Event-Based Systems (DEBS'08)*. ACM Press, New York, 83–94.

Ferretti, S. and Roccetti, M. 2006. AC/DC: An algorithm for cheating detection by cheating. In *Proceedings of the International Workshop on Network and Operating Systems Support for Digital audio and Video (NOSSDAV'06)*. ACM Press, New York.

FFXI. 2011. Final fantasy XI. http://www.playonline.com/ff11us.

Floyd, S. and Jacobson, V. 1993. Random early detection gateways for congestion avoidance. *IEEE Trans. Netw. 1*, 4, 397–413.

FORD, B., SRISURESH, P., AND KEGEL, D. 2005. Peer-to-peer communication across network address translators. In *Proceedings of the USENIX Annual Technical Conference (ATC'05).* USENIX, 179–192.

FRECON, E. AND STENIUS, M. 1998. DIVE: A scalable network architecture for distributed virtual environments. *Distrib. Syst. Engin. J. 5*, 3, 91–100.

FREY, D., ROYAN, J., PIEGAY, R., KERMARREC, A. M., ANCEAUME, E., AND LE FESSANT, F. 2008. Solipsis: A decentralized architecture for virtual environments. In *Proceedings of the International Workshop on Massively Multiuser Virtual Environments (MMVE'08).*

FUJIMOTO, R. 2000. *Parallel and Distributed Simulation Systems*. Wiley Interscience.

FUNG, Y. S. 2006. Hack-proof synchronization protocol for multi-player online games. In *Proceedings of the ACM SIGCOMM International Workshop on Network and System Support for Games (NETGAMES'06).*

GAIKAI 2011. Gaikai video game advertising network. http://www.gaikai.com.

GAMERANK 2011. GameRank. http://us.playstation.com/psn/.

GAO, L., DAHLIN, M., NAYATE, A., ZHENG, J., AND IYENGAR, A. 2003. Application specific data replication for edge services. In *Proceedings of the International Conference on World Wide Web (WWW'03).* ACM Press, New York, 449.

GAUTHIERDICKEY, C., LO, V., AND ZAPPALA, D. 2005. Using n-trees for scalable event ordering in peer-to-peer games. In *Proceedings of the International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV'05).* ACM Press, New York, 87.

GAUTHIERDICKEY, C., ZAPPALA, D., LO, V., AND MARR, J. 2004. Low latency and cheat-proof event ordering for peer-to-peer games. In *Proceedings of the International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV'04).* ACM Press, New York, 134–139.

GAUTIER, L., DIOT, C., AND KUROSE, J. 1999. End-to-end transmission control mechanisms for multiparty interactive applications on the internet. In *Proceedings of the IEEE International Conference on Computer Communications (INFOCOM'99).* Vol. 3.

GILBERT, S. AND LYNCH, N. 2002. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News 33*, 2, 59.

GILMORE, J. AND ENGELBRECHT, H. 2011. A survey of state persistency in peer-to-peer massively multiplayer online games. *IEEE Trans. Parallel Distrib. Syst. 23,* 5, 818–834.

GOLLE, P. AND DUCHENEAUT, N. 2005. Preventing bots from playing online games. *Comput. Entertain. 3*, 3, 3.

GOODMAN, J. 2008. A hybrid design for cheat detection in massively multiplayer online games. M.S. thesis, McGill University. http://gram.cs.mcgill.ca/theses/goodman-08-hybrid.pdf.

GOODMAN, J. AND VERBRUGGE, C. 2008. A peer auditing scheme for cheat elimination in mmogs. In *Proceedings of the International ACM SIGCOMM Workshop on Network and System Support for Games (NETGAMES'08).*

GREENHALGH, C. AND BENFORD, S. 2002. MASSIVE: A distributed virtual reality system incorporating spatial trading. In *Proceedings of the International Conference on Distributed Computing Systems (ICDCS'02).* 27–34.

GUERRAOUI, R., HUGUENIN, K., KERMARREC, A.-M., MONOD, M., AND PRUSTY, S. 2010. LiFTinG: Lightweight protocol for free-rider-tracking in gossip. In *Proceedings of International Middleware Conference (Middleware'10).* ACM/IFIP/USENIX.

GUPTA, M., JUDGE, P., AND AMMAR, M. 2003. A reputation system for peer-to-peer networks. In *Proceedings of the International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV'03).* ACM Press, New York, 144–152.

HAEBERLEN, A., ADITYA, P., RODRIGUES, R., AND DRUSCHEL, P. 2010. Accountable virtual machines. In *Proceedings of the International Conference on Operating Systems Design and Implementation (OSDI'10).* USENIX, 1–16.

HAMILTON, J. A., NASH, D. A., AND POOCH, U. W. 1997. *Distributed Simulation*. CRC Press, Boca Raton, FL.

HAMPEL, T., BOPP, T., AND HINN, R. 2006. A peer-to-peer architecture for massive multiplayer online games. In *Proceedings of the International ACM SIGCOMM Workshop on Network and System Support for Games (NETGAMES'06).*

HARVEY, R. C., HAMZA, A., LY, C., AND HEFEEDA, M. 2010. Energy-efficient gaming on mobile devices using dead reckoning-based power management. In *Proceedings of the International ACM SIGCOMM Workshop on Network and System Support for Games (NETGAMES'10).* 4.

HENNING, M. 2004. Massively multiplayer middleware. *Queue Mag. 1*, 10, 38–45.

HOGLUND, G. AND MCGRAW, G. 2007. *Exploiting Online Games: Cheating Massively Distributed Systems* 1st Ed. Addison-Wesley.

HU, S. Y., CHANG, S. C., AND JIANG, J. R. 2008. Voronoi state management for peer-to-peer massively multiplayer online games. In *Proceedings of the International IEEE Conference on Consumer Communications and Networking (CCNC'08).* 1134–1138.

Hu, S. Y., Chen, J. F., and Chen, T. H. 2006. VON: A scalable peer-to-peer network for virtual environments. *IEEE Netw. 20*, 4, 22–31.

Hu, S. Y., Jiang, J. R., and Chen, B. Y. 2010. Peer-to-peer 3d streaming. *IEEE Internet Comput. 14*, 2, 54–61.

Hu, S. Y. and Liao, G. M. 2004. Scalable peer-to-peer networked virtual environment. In *Proceedings of the International ACM SIGCOMM Workshop on Network and System Support for Games (NETGAMES'04)*. 129–133.

Huang, G. Y., Hu, S. Y., and Jiang, J. R. 2008a. Scalable reputation management for p2p mmogs. In *Proceedings of the International Workshop on Massively Multiuser Virtual Environments (MMVE'08)*. 1–5.

Huang, G. Y., Hu, S. Y., and Jiang, J. R. 2008b. Scalable reputation management with trustworthy user selection for p2p mmogs. *Proc. Int. J. Adv. Media Comm. 2*, 380–401.

Huguenin, K., Yahyavi, A., and Kemme, B. 2011a. Cheat detection and prevention in p2p mmogs. Tech. rep. SOCS-TR-2011.5, McGill University, Canada.

Huguenin, K., Yahyavi, A., and Kemme, B. 2011b. Cheat detection and prevention in p2p mogs. In *Proceedings of the International ACM SIGCOMM Workshop on Network and System Support for Games (NETGAMES'11)*. 1–2.

IEEE. 1995. IEEE standard for distributed interactive simulation application protocols, IEEE standard 1278.1-1995. Tech. rep., IEEE.

IEEE. 2000. IEEE standard for modelling and simulation high level architecture (hla). Tech. rep., IEEE. http://standards.ieee.org/findstds/standard/1516-2000.html.

Iimura, T., Hazeyama, H., and Kadobayashi, Y. 2004. Zoned federation of game servers: A peer-to-peer approach to scalable multi-player online games. In *Proceedings of the International ACM SIGCOMM Workshop on Network and System Support for Games (NETGAMES'04)*. ACM Press, New York, 116–120.

Iosup, A., Lascateu, A., and Tapus, N. 2010. CAMEO: enabling social networks for massively multiplayer online games through continuous analytics and cloud computing. In *Proceedings of the International ACM SIGCOMM Workshop on Network and System Support for Games (NETGAMES'10)*. 7.

Izaiku, T., Yamamoto, S., Murata, Y., Shibata, N., Yasumoto, K., and Ito, M. 2006. Cheat detection for mmorpg on p2p environments. In *Proceedings of the International ACM SIGCOMM Workshop on Network and System Support for Games (NETGAMES'06)*.

Jaramillo, J. E., Escobar, L., and Trefftz, H. 2003. Area of interest management by grid-based discrete aura approximations for distributed virtual environments. In *Proceedings of the International Symposium on Virtual Reality (SVR'03)*. 342–353.

Jefferson, D. R. 1985. Virtual time. *ACM Trans. Programm. Lang. Syst. 7*, 3, 404–425.

Jelasity, M., Montresor, A., and Babaoglu, O. 2009. T-Man: Gossip-based fast overlay topology construction. *Comput. Netw. 53*, 2321–2339.

Kabus, P., Terpstra, W., Cilia, M., and Buchmann, A. 2005. Addressing cheating in distributed MMOGs. In *Proceedings of the International ACM SIGCOMM Workshop on Network and System Support for Games (NETGAMES'05)*.

Kaiser, E. and Feng, W. C. 2009. PlayerRating: A reputation system for multiplayer online games. In *Proceedings of the International ACM SIGCOMM Workshop on Network and System Support for Games (NETGAMES'09)*. 8:1–8:6.

Kamvar, S., Schlosser, M., and Garcia-Molina, H. 2003. The eigentrust algorithm for reputation management in p2p networks. In *Proceedings of the International Conference on World Wide Web (WWW'03)*.

Kawahara, Y., Aoyama, T., and Morikawa, H. 2004. A peer-to-peer message exchange scheme for large-scale networked virtual environments. *Telecomm. Syst. 25*, 3, 353–370.

Keller, J. and Simon, G. 2003. Solipsis: A massively multi-participant virtual world. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'03)*.

Kienzle, J., Verbrugge, C., B, K., Denault, A., and Hawker, M. 2009. Mammoth: A massively multiplayer game research framework. In *Proceedings of the International Conference on Foundations of Digital Games (FDG'09)*. ACM Press, New York, 308–315.

Knutsson, B., Lu, H., Xu, W., and Hopkins, B. 2004. Peer-to-peer support for massively multiplayer games. In *Proceedings of the IEEE International Conference on Computer Communications (INFOCOM'04)*.

Korhonen, H. and Koivisto, E. M. I. 2007. Playability heuristics for mobile multi-player games. In *Proceedings of the International Conference on Digital Interactive Media in Entertainment and Arts (DIMEA'07)*. ACM Press, New York, 28–35.

Kulkarni, S., Douglas, S., and Churchill, D. 2010. Badumna: A decentralised network engine for virtual environments. *Comput. Netw. 54*, 12, 1953–1967.

LAURENS, P., PAIGE, R., BROOKE, P., AND CHIVERS, H. 2007. A novel approach to the detection of cheating in multiplayer online games. In *Proceedings of the International Conference on Engineering of Complex Computer Systems (ICECCS'07).*

LETY, E., GAUTIER, L., AND DIOT, C. 1998. MiMaze, a 3d multi-player game on the internet. In *Proceedings of the International Conference on Virtual Systems and Multimedia.*

LETY, E., TURLETTI, T., AND BACCELLI, F. 2004. SCORE: A scalable communication protocol for large-scale virtual environments. *IEEE Trans. Netw. 12*, 2, 247–260.

LI, H. C., CLEMENT, A., MARCHETTI, M., KAPRITSOS, M., ROBISON, L., ALVISI, L., AND DAHLIN, M. 2008. FlightPath: Obedience vs. choice in cooperative services. In *Proceedings of the International Conference on Operating Systems Design and Implementation (OSDI'08).* USENIX, 355–368.

LI, K., DING, S., MCCREARY, D., AND WEBB, S. 2004. Analysis of state exposure control to prevent cheating in online games. In *Proceedings of the International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV'04).* ACM Press, New York, 140–145.

LIU, H. I. AND LO, Y. T. 2008. DaCAP - A distributed anti-cheating peer to peer architecture for massive multiplayer on-line role playing game. In *Proceedings of the International Symposium on Cluster Computing and the Grid (CCGRID'08).* IEEE/ACM, 584–589.

MACEDONIA, M. R., ZYDA, M. J., PRATT, D. R., BRUTZMAN, D. P., AND BARHAM, P. 1995. Exploiting reality with multicast groups. *IEEE Comput. Graph. Appl. 15*, 5, 38–45.

MAKBILY, Y., GOTSMAN, C., AND BAR-YEHUDA, R. 1999. Geometric algorithms for message filtering in decentralized virtual environments. In *Proceedings of the ACM Symposium on Interactive 3D Graphics (ACMI3D'99).* 39–46.

MAUVE, M., VOGEL, J., HILT, V., AND EFFELSBERG, W. 2004. Local-lag and timewarp: providing consistency for replicated continuous applications. *IEEE Trans. Multimedia 6*, 47–57.

MCCOY, A., MCLOONE, S., WARD, T., AND DELANEY, D. 2005. Dynamic hybrid strategy models for networked multiplayer games. In *Proceedings of International European Conference on Modelling and Simulation (ECMS'05).* 727–732.

MCCOY, A., WARD, T., MCLOONE, S., AND DELANEY, D. 2007. Multistep-ahead neural-network predictors for network traffic reduction in distributed interactive applications. *ACM Trans. Model. Comput. Simul. 17,* 1–30.

MILLER, J. L. AND CROWCROFT, J. 2010. The near-term feasibility of p2p mmogs. In *Proceedings of the International ACM SIGCOMM Workshop on Network and System Support for Games (NETGAMES'10).*

MOGAKI, S., KAMADA, M., YONEKURA, T., OKAMOTO, S., OHTAKI, Y., AND REAZ, M. 2007. Time-stamp service makes real-time gaming cheat-free. In *Proceedings of the International ACM SIGCOMM Workshop on Network and System Support for Games (NETGAMES'07).*

MONCH, C., GRIMEN, G., AND MIDTSTRAUM, R. 2006. Protecting online games against cheating. In *Proceedings of the International ACM SIGCOMM Workshop on Network and System Support for Games (NETGAMES'06).*

MORALES, R. AND GUPTA, I. 2009. AVMON: Optimal and scalable discovery of consistent availability monitoring overlays for distributed systems. *IEEE Trans. Parallel Distrib. Syst. 20*, 446–459.

MORSE, K. L. 1996. Interest management in large-scale distributed simulations. Tech. rep. ICS-TR-96-27. http://www.cs.bham.ac.uk/research/projects/pdesmas/LITERATURE/rzm/hla/morse:96.pdf.

MULLIGAN, J., PATROVSKY, B., AND KOSTER, R. 2003. *Developing Online Games: An Insider's Guide.* Pearson Education.

NAGEL, W., WALTER, W., LEHNER, W., LEONTIADIS, E., DIMAKOPOULOS, V., AND PITOURA, E. 2006. Creating and maintaining replicas in unstructured peer-to-peer systems. In *Proceedings of the International European Conference on Parallel Processing (EuroPar'06).* Vol. 4128. Springer, 1015–1025.

NAJARAN, M. T. AND KRASIC, C. 2010. Scaling online games with adaptive interest management in the cloud. In *Proceedings of the International ACM SIGCOMM Workshop on Network and System Support for Games (NETGAMES'10).*

NINTENDO. 2011. DS at nintendo. http://www.nintendo.com/ds.

ONLIVE. 2011. OnLive. http://www.onlive.com.

OTTO, J. S., SÁNCHEZ, M. A., CHOFFNES, D. R., BUSTAMANTE, F., AND SIGANOS, G. 2011. On blind mice and the elephant: Understanding the network impact of a large distributed system. In *Proceedings of the ACM SIGCOMM Conference*. 110–121.

OZSU, M. T. AND VALDURIEZ, P. 2011. *Principles of Distributed Database Systems*. Springer.

PALAZZI, C. E., FERRETTI, S., CACCIAGUERRA, S., AND ROCCETTI, M. 2006. Interactivity-loss avoidance in event delivery synchronization for mirrored game architectures. *IEEE Trans. Multimedia 8*, 4, 874–879.

PANTEL, L. AND WOLF, L. 2002a. On the suitability of dead reckoning schemes for games. In *Proceedings of the International ACM SIGCOMM Workshop on Network and System Support for Games (NETGAMES'02).*

PANTEL, L. AND WOLF, L. C. 2002b. On the impact of delay on real-time multiplayer games. In *Proceedings of the International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV'02).* 23–29.

PITTMAN, D. AND GAUTHIERDICKEY, C. 2007. A measurement study of virtual populations in massively multiplayer online games. In *Proceedings of the International ACM SIGCOMM Workshop on Network and System Support for Games (NETGAMES'07).* 25–30.

PITTMAN, D. AND GAUTHIERDICKEY, C. 2011. Cheat-proof peer-to-peer trading card games. In *Proceedings of the International ACM SIGCOMM Workshop on Network and System Support for Games (NETGAMES'11).*

PSN. 2011. Playstation network PSN. http://us.playstation.com/psn/.

PSP. 2011. PlayStation portable. http://us.playstation.com/psp.

PUNKBUSTER. 2011. PunkBuster: The original anti-cheat system for online multiplayer games. http://www.evenbalance.com/.

QUAZAL 2011. Quazal. http://www.quazal.com/en/products/net-z.

ROSENBERG, J., WEINBERGER, J., HUITEMA, C., AND MAHY, R. 2003. STUN - Simple traversal of user datagram protocol (udp) through network address translators (nats). http://www.ietf.org/rfc/rfc3489.txt.

ROWSTRON, A. AND DRUSCHEL, P. 2001a. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Proceedings of the International Middleware Conference (Middleware'01).* ACM/IFIP/USENIX, 329–350.

ROWSTRON, A. AND DRUSCHEL, P. 2001b. Storage management and caching in past, a large-scale, persistent peer-to-peer storage utility. *SIGOPS Oper. Syst. Rev. 35*, 188–201.

SCHLUESSLER, T., GOGLIN, S., AND JOHNSON, E. 2007. Is a bot at the controls? Detecting input data attacks. In *Proceedings of the International ACM SIGCOMM Workshop on Network and System Support for Games (NETGAMES'07).* 1–6.

SCHMIEG, A., STIELER, M., JECKEL, S., KABUS, P., KEMME, B., AND BUCHMANN, A. 2008. pSense - Maintaining a dynamic localized peer-to-peer structure for position based multicast in games. In *Proceedings of the International Conference on Peer-to-Peer Computing (P2P'08).*

SEAH, D., LEONG, W. K., YANG, Q., LEONG, B., AND RAZEEN, A. 2009. Peer nat proxies for peer-to-peer games. In *Proceedings of the International ACM SIGCOMM Workshop on Network and System Support for Games (NETGAMES'09).* 6:1–6.

SECONDLIFE. 2011. Second life: Virtual worlds, Avatars, free 3D chat, online meetings. http://secondlife.com.

SHELDON, N., GIRARD, E., BORG, S., CLAYPOOL, M., AND AGU, E. 2003. The effect of latency on user performance in warcraft III. In *Proceedings of the International ACM SIGCOMM Workshop on Network and System Support for Games (NETGAMES'03).* 3–14.

SHOWEQ. 2011. Showeq. http://www.showeq.net.

SOZIO, M., NEUMANN, T., AND WEIKUM, G. 2008. Near-optimal dynamic replication in unstructured peer-to-peer networks. In *Proceedings of the ACM International Symposium on Principles of Database Systems (PODS'08).* 281–290.

STOICA, I., MORRIS, R., KARGER, D., KAASHOEK, M. F., AND BALAKRISHNAN, H. 2001. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the ACM SIGCOMM Conference.* 149–160.

SUNDARESAN, S., DE DONATO, W., FEAMSTER, N., TEIXEIRA, R., CRAWFORD, S., AND PESCAPE, A. 2011. Broadband internet performance: A view from the gateway. In *Proceedings of the ACM SIGCOMM Conference.* 134–145.

SUZNJEVIC, M., DOBRIJEVIC, O., AND MATIJASEVIC, M. 2009. MMORPG player actions: Network performance, session patterns and latency requirements analysis. *Multimedia Tools Appl. 45*, 1, 191–214.

SUZNJEVIC, M. AND MATIJASEVIC, M. 2012. Player behavior and traffic characterization for MMORPGs: A survey. *Multimedia Syst. 2013,* 199–220.

SUZNJEVIC, M., STUPAR, I., AND MATIJASEVIC, M. 2011. Traffic modeling of player action categories in a mmorpg. In *Proceedings of the International ICST Conference on Simulation Tools and Techniques (SIMUTools'11).* 280–287.

SWAMYNATHAN, G., ALMEROTH, K. C., AND ZHAO, B. Y. 2010. The design of a reliable reputation system. *Electron. Commerce Res. 10*, 239–270.

SWAMYNATHAN, G., ZHAO, B. Y., AND ALMEROTH, K. C. 2008. Exploring the feasibility of proactive reputations. *Concurrency Comput. Pract. Exper. 20,* 155–166.

TANIN, E., HARWOOD, A., AND SAMET, H. 2007. Using a distributed quadtree index in peer-to-peer networks. In *Proceedings of the International Conference on Very Large Data Bases (VLDB'07).* Vol. 16. 165–178.

TERAZONA. 2002. Terazona: Zona application frame work white paper. http://www.zona.net/whitepaper/Zonawhitepaper.pdf.

TERRY, D. B., THEIMER, M. M., PETERSEN, K., DEMERS, A. J., SPREITZER, M. J., AND HAUSER, C. H. 1995. Managing update conflicts in bayou, a weakly connected replicated storage system. In *Proceedings of the ACM Symposium on Operating System Principles (SOSP'95).* 172–182.

VAC. 2011. VAC: Valve anti cheat. https://support.steampowered.com/kb_article.php?p_faqid=370.

VARVELLO, M., DIOUT, C., AND BIERSACK, E. 2009a. P2P Second life: Experimental validation using Kad. In *Proceedings of the IEEE International Conference on Computer Communications (INFOCOM'09).* 1161–1169.

VARVELLO, M., FERRARI, S., BIERSACK, E., AND DIOT, C. 2009b. Distributed avatar management for second life. In *Proceedings of the International ACM SIGCOMM Workshop on Network and System Support for Games (NETGAMES'09).* 5:1–5:6.

VARVELLO, M., FERRARI, S., BIERSACK, E., AND DIOT, C. 2011. Exploring second life. *IEEE Trans. Netw. 19,* 1, 80–91.

VARVELLO, M. AND VOELKER, G. M. 2010. Second life: A social network of humans and bots. In *Proceedings of the International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV'10).* 9–14.

WANG, X., KIM, H., VASILAKOS, A. V., KWON, T. T., CHOI, Y., CHOI, S., AND JANG, H. 2009. Measurement and analysis of world of warcraft in mobile wimax networks. In *Proceedings of the International ACM SIGCOMM Workshop on Network and System Support for Games (NETGAMES'09).*

WEBB, S., SOH, S., AND LAU, W. 2007. RACS: A referee anti-cheat scheme for p2p gaming. In *Proceedings of the International Workshop on Network the Operating Systems Support for Digital Audio and Video (NOSSDAV'07).*

WEBB, S. D. AND SOH, S. 2007. Cheating in networked computer games: a review. In *Proceedings of the International Conference on Digital Interactive Media in Entertainment and Arts (DIMEA'07).* 105–112.

WEBB, S. D., SOH, S., AND TRAHAN, J. L. 2009. Secure referee selection for fair and responsive peer-to-peer gaming. *Simul. 85*, 608–618.

WIERZBICKI, A. AND KASZUBA, T. 2007. Practical trust management without reputation in peer-to-peer games. *Multiagent Grid Syst. 3*, 411–428.

WU, C. H., HU, S. Y., AND TSENG, L. M. 2009. Discovery of physical neighbors for p2p 3d streaming. In *Proceedings of International Conference on Ultra Modern Telecommunications Workshops (ICUMT'09).* 1–6.

XBOX. 2011. XBox Live. http://www.xbox.com/live.

YAHYAVI, A., HUGUENIN, K., AND KEMME, B. 2011. AntReckoning: A pheromone-based dead reckoning algorithm for games. In *Proceedings of the International ACM SIGCOMM Workshop on Network and System Support for Games (NETGAMES'11).*

YAHYAVI, A., HUGUENIN, K., AND KEMME, B. 2012. Interest modeling in games: The case of dead reckoning. *Multimedia Syst. 19,* 3, 255–270.

YAMAMOTO, S., MURATA, Y., YASUMOTO, K., AND ITO, M. 2005. A distributed event delivery method with load balancing for mmorpg. In *Proceedings of the International ACM SIGCOMM Workshop on Network and System Support for Games (NETGAMES'05).*

YAN, J. AND RANDELL, B. 2005. A systematic classification of cheating in online games. In *Proceedings of the International ACM SIGCOMM Workshop on Network and System Support for Games (NETGAMES'05).* 1–9.

YU, A. AND VUONG, S. T. 2005. MOPAR: A mobile peer-to-peer overlay architecture for interest management of massively multiplayer online games. In *Proceedings of the International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV'05).*

ZHANG, K. 2010. Persistent transaction models for massively multiplayer online games. M.S. thesis, McGill University.

ZHANG, K. AND KEMME, B. 2011. Transaction models for massively multiplayer online games. In *Proceedings of the International Symposium on Reliable Distributed Systems (SRDS'11).* 31–40.

ZHAO, B. Y., KUBIATOWICZ, J. D., AND JOSEPH, A. D. 2001. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Tech. rep. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.111.1818&rep=rep1&type=pdf.

ZYNGA. 2011. Zynga - Connecting the world through games. http://www.zynga.com/games.

Received April 2011; revised September 2012; accepted January 2013