Watchmen: Scalable Cheat-Resistant Support for Distributed Multi-Player Online Games

Amir Yahyavi*, Kévin Huguenin[†], Julien Gascon-Samson*, Jörg Kienzle* and Bettina Kemme*
*McGill University, Montreal, Quebec, Canada
[†]EPFL, Lausanne, Switzerland

***Abstract*—Multi-player online games are inherently distributed applications, and a wide range of distributed architectures have been proposed. However, only few successful commercial systems follow such approaches, even given their benefits, due to one main hurdle: the easiness with which cheaters can disrupt the game state computation and dissemination, perform illegal actions, or unduly gain access to sensitive information. The challenge is that any measures used to address cheating must meet the heavy scalability and tight latency requirements of fast paced games.**

We propose Watchmen, the first distributed scalable protocol designed with cheat detection and prevention in mind that supports fast paced games. It is based on a randomized dynamic proxy scheme for both the dissemination and verification of actions. Furthermore, Watchmen reduces the information exposed to players close to the minimum required to render the game. We build our proof-of-concept prototype on top of Quake III. We show that Watchmen, while scaling to hundreds of players and meeting the tight latency requirements of first person shooter games, is able to significantly reduce opportunities to cheat, even in the presence of collusion.

I. Introduction

While massively multi-player games such as World of Warcraft are the best known genre of large-scale gaming environments, fast-paced multi-player games, mostly *First-Person Shooter* (FPS) games such as Quake and Halo, have been popular for decades. In the traditional setting, a small number of players (e.g, up to 16) play relatively short instances of the game on a local area network following a client/server architecture, where one of the player machines hosts the server, which controls all game actions and communication. In such a case, the players are typically friends and thus, cheating is not a critical issue. However, this genre of games has evolved quickly over the years. Popular game networks (e.g., XBox Live, PSN) and the concept of game lobbies [1, 2] allows players across the world to connect and participate together in such games. Thus, there is no more a homogeneous local area network that would connect the players, and the trust and solidarity that used to exist among players has vanished. Furthermore, as it has become so easy to find a large set of players that want to play together at the same time, the desire arises to develop a new generation of this genre where hundreds of players can play within the same game instance. Therefore, hosting the server on one of the client machines has become infeasible: such a client would have many powerful options for cheating and the client machines are unlikely to handle the game server load in terms of processing power and upload bandwidth.

One approach to support such games would be to transfer the game server into the cloud. This is cheaper for the gaming company than hosting large and expensive custom built server farms themselves, because the system can scale dynamically with the number of isolated concurrent game instances. However, several problems arise: (1) the costs are still non-negligible, (2) single cloud instances may not scale to hundreds of players in the same game world, thus, requiring distributed solutions that incur higher costs and complexity, (3) these solutions do not support spontaneous, short-lived, and locally run gaming sessions particularly by third parties or gamenets. Thus, the question arises why not keep the game execution with the clients and develop a decentralized solution.

Over the last years, several decentralized solutions have been developed (e.g. [3–5]). They distribute the load among the players, and gain scalability by introducing new resources with every joining player. However, many of them ignore cheating, which is a serious threat as players have access to and can manipulate sensitive game data [6]. The decentralized approaches that address cheating have typically been developed for strategy games (e.g., lockstep [7]) and are usually not efficient enough for fast-paced games.

Cheating essentially consists of gaining an unfair advantage and comes mostly in the following three forms: disrupting the game state computation and dissemination, performing illegal actions, and gaining access to sensitive information [8]. In server-based game engines, cheat detection and prevention can be achieved by making the server verify the players' actions, ensure synchronization, and reduce the information sent to players to the minimal amount required to render the game world [7]. Nevertheless, even server-based systems are vulnerable to cheats, particularly when a player becomes the server. In decentralized games, detecting cheating is even more difficult and natural trade-offs between responsiveness, scalability, verification and information disclosure have to be made. One has to be aware that while security is important in games, it often comes second to performance. While millions of players currently play games that provide no or low security measures[1,2], only a few would play a very secure but low-performance game.

Motivated by these developments, this paper presents Watchmen, a distributed, scalable, and cheat-resistant architecture designed for FPS games. Our goal is to design a platform that is *reasonably* secure and able to handle fast paced FPS games. Specifically, we aim to (1) provide a low-latency distributed infrastructure; (2) not rely on a central server or trusted third parties but be able to take advantage of them should they be present; (3) limit the opportunities for cheating even in the presence of collusion; (4) detect cheaters during game play. To achieve this Watchmen uses a combination of both novel and well-established techniques:

- *Randomization:* player connections are randomized and frequently changed to avoid unfairness, collusion, and other forms of cheating.
- *Cross verification:* requests and updates are cross verified by players to ensure validity and correctness.
- *Information hiding:* information available to players is kept close to the minimum required to render the game.
- *Multi resolution:* players request and receive updates at different rates depending on the current situation.

A major challenge addressed by Watchmen is finding the right balance between the following contradictory goals: meet the tight scalability and latency constraints, provide enough information to players about each other to allow efficient mutual verifications, and at the same time limit this information to the minimum to limit cheating opportunities.

We chose the popular open-source FPS game Quake III to implement and evaluate our system, and have extended the platform to support up to 48 players in a decentralized setting. The reason for using Quake III is the game's popularity and the extensive past research conducted on it, which makes it easier to determine the base game requirements and to compare our results. We show that Watchmen scales far beyond the standard number of players supported by Quake. We also show that opportunities to cheat are significantly reduced, even in the presence of collusion (often ignored), while keeping good performance with respect to scalability and responsiveness. Although the paper focuses on first person shooter games, we believe that the concepts and mechanisms used in Watchmen can be applied to a broad range of games including role playing and real-time strategy games, as they have lower networking requirements [4].

II. Background

A. Multi-player on-line games

In multi-player on-line games, players experience the action through a character they control, referred to as *avatar*. Avatars evolve in a virtual *game world* and can interact with objects and other avatars, controlled by other players or by artificial intelligence. The state of an avatar typically includes its position, aim, objects it owns, health, *etc.* and is modified by the instructions of the player (e.g., move or shoot) and the interactions with other avatars or objects (e.g., collision). To visualize the game world on the screen, a player needs (at least partial) *replicas* of other avatars and game objects that are in its own avatar's vision field. If the state of an avatar changes, update messages must be sent to those players that need this information. State updates account for the largest part of the bandwidth needs of MOGs.

Games usually run in a discrete event-loop, meaning that in each *frame* the states of the entities are updated and updates may be sent. The frame rate is crucial in the design of MOGs. In Quake III, each frame is 50 ms. Given the short duration of each frame, updates show high temporal similarities and can be *delta-coded*, only including the differences between updates.

Latency Requirements: Games have tight *latency* requirements and *lag*, here defined as the difference between the games state at the player and the actual state, has an adverse effect on the game-play. In Quake III and similar games, latencies and lags of up to 150 ms are tolerable [9–12]. To meet these requirements, games limit the geographic location of players to the same country or continent [2, 13]. In addition, they rely on UDP for faster communication.

Decentralization: In a centralized game, players send all their updates to the server that controls all game entities. The server verifies each update and sends it, given its global knowledge, to only those players who need it (if designed well). In a decentralized system, players typically control their own avatars. Bots and game items can be distributed across the players as well or be controlled by the company's servers. In small to medium sized games (e.g., FPS games such as Quake III), players are usually aware of all entities of the game. In large-scale RPGs (e.g., World of Warcraft) decentralization is achieved by dividing the game world into zones (e.g., [14]). A simple peer-to-peer game lets each player send all changes directly to all other players. However, as players have limited bandwidth, P2P MOGs use a range of techniques to increase scalability, which we describe in the following section.

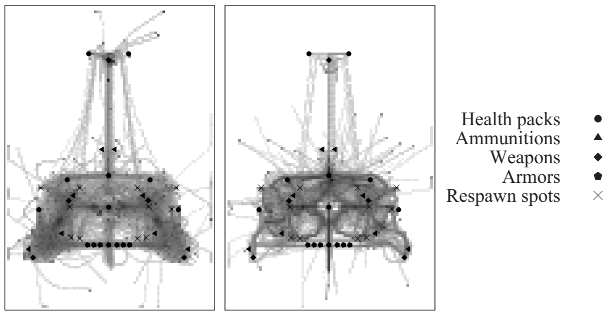
Bandwidth Requirements: Most broadband connections are asymmetric, with upload bandwidth being the limitation. In a distributed scenario where players all send updates to each other, the growth in overall bandwidth requirements is *quadratic*. Average bandwidth requirements in centralized Quake III is $12n$ kbps where n is the number of players [5].

B. Interest Filtering

Interest filtering is designed to limit the updates a player sends; for example, only to entities inside a fixed-radius sphere around the avatar. In Quake III, this is done via

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

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



(a) Human movements (b) NPC Movements

Figure 1. Heatmap of player positions in a Quake III *deathmatch* game in the `q3dm17` map. Darker colors show higher presence in a region.

potentially visible sets (PVS) that determine which players are visible and hence should receive an update. However, using a fixed geographical area as *Area-of-Interest* (AOI) does not necessarily bound the actual number of players inside the area. As Figure 1 shows, players show an exponential presence in some area of the game, due to their strategic location or presence of important game items, rendering AOI filtering unusable (color intensity is normalized logarithmic values of presence in each region). Non-player characters (NPC), i.e., bots, further worsen the situation as they tend to use predetermined paths and locations.

Variable object fidelity: To address this issue games use a multi-resolution schemes where updates are sent at different rates to players. Multi-resolution selective dissemination schemes have been used in distributed simulations as well as virtual environments and games. Gaming engines exist, e.g., BigWorld [15], where players receive different *Level of Details* (LoD) based on a metric determined by the games. Donnybrook, designed for FPS games, instead uses the set of the top 5 avatars with respect to an attention metric based on proximity, aim and interaction recency, called interest set (IS). A player typically receives frequent updates only about avatars in his IS and infrequent so-called *dead-reckoning* updates (see below) about other avatars which contain information to simulate the expected motion of the avatar for several frames.

Dead reckoning: Is the process of predicting the state of an avatar based on past observations, thus allowing to reduce the frequency of position updates while keeping the display smooth [16]. Players who do not send frequent position updates rely on infrequent dead-reckoning messages containing the avatar's expected next position and aim (computed locally) and its current position, aim, rate of fire, *etc.*

III. The Watchmen Architecture

Watchmen aims to provide security measures while being able to handle large-scale fast-paced games. The goal is to substantially reduce the potential for cheating. In general, cheating can be roughly divided into three categories: (1) *Disruption of information flow* covers actions that stop or change the normal pace of information flow (e.g., dropping messages), (2) *Invalid updates* cover actions that are invalid according to game rules (e.g., too fast moves), repetitions, or spoofing, (3) *Unauthorized access* includes any action (e.g., sniffing) that enables access to unauthorized information. In Table I, we describe a range of cheating types within each category (for details see [14]).

As we discuss our architecture, we describe how Watchmen detects or prevents these cheat types. Note that there is a range of cheat detection and prevention techniques that can be applied to any game architecture including Watchmen (e.g., code tampering detection software or hardware, aimbot detection tools, CAPTCHAs, tamper-resistant logging). In our discussion, we focus on security mechanisms that are based on the specific architecture of Watchmen. Moreover, while our focus is on gaming, techniques presented can be used in fault detection and dealing with selfish and malicious nodes in other similar distributed systems.

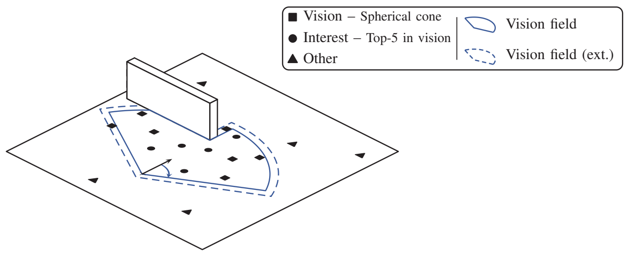


Figure 2. Subscription-types and corresponding areas: Vision set (VS) is composed of avatars inside a fixed-radius ($\pm$60 degrees) and angle spherical cone directed along the player's aim; Interest set (IS) is composed of the 5 avatars inside VS which catch the player's attention the most.

Watchmen achieves its goal through three main techniques (we justify our choices in the security analysis): (1) with *vision-based information filtering* (Figure 2), players receive updates at different rates and containing different information depending on the players' location in the game and his surroundings. To do this, players subscribe only to the information they should receive. (2) *Proxy-based indirect communication* (Figure 3), where each player is periodically assigned a new random proxy responsible for a player's update dissemination and subscription messages. (3) *Mutual verification* verifying players' updates, actions, and subscriptions.

A. Subscription Model

Watchmen uses the notion of *interest*. Players receive updates at different rates and containing different information according to the details required to render the game (their interests). This concept has been successfully used in game engines and has been shown to greatly increase scalability [5] while maintaining game experience. However, in our context it helps us to reduce the chances for unauthorized access to sensitive information as we limit the information available to a player as much as possible.

Table I
POPULAR CHEATING MECHANISMS IN DISTRIBUTED MULTI-PLAYER GAMES

Type	Name	Description	Watchmen
Disruption of information flow	Escaping	Terminating the connection in order to escape imminent loss	Detected by proxy and others
	Time cheating (look ahead)	Delaying the updates to base one's actions on those received from others	Detected by proxy and others
	Network flooding	Overflowing the game server to create lags and disrupt game play	Prevented through distribution
	Fast rate cheat	Mimicking a rate of game event generation that is faster than the real one	Detected by proxy and others
	Suppress-correct cheat	Dropping consecutive updates, then sending an invalid update afterwards	Detected by proxy and others
	Replay cheat	Resend signed & encrypted updates of a different player	Prevented/Detected by proxy and others
	Blind opponent	Dropping updates to opponents, blinding them about the cheater's actions	Detected by proxy and others
Invalid updates	Client-side code tampering	Modifying the client-side code to get an unfair advantage	Detected by sanity checks & action repetition
	Aimbots	Using an intelligent program to provide it with automatic weapon aiming	Detection by proxy (statistical analysis)
	Spoofing	Sending messages, pretending to be a different player	Detected by players
	Consistency cheat	Sending different updates to different players	Prevented by proxy and others
Unauthorized access	Sniffing	Logging & accessing different information sent across the network	Prevented by minimizing information exposure
	Maphack	Hacking to see through walls and obstacles	Prevented by minimizing information exposure
	Rate Analysis	Analyzing the updates rates to detect players attention and escape	Prevented by proxy and subscription model

The technique works as follows: each player partitions the other players in the game into 3 different sets as shown in Figure 2. The player sends subscription requests to the other players according to what set they belong to and receive 3 types of updates according to their subscription:

Vision Set (VS): this set contains all the avatars visible to the player. In FPS games, it corresponds to a spherical cone, defined by a given radius and angle (e.g., ± 60 degrees in Quake III) and centered on the avatar. In practice, the cone is made slightly larger than the actual avatar's vision field (see Figure 2) in order to handle fast avatar movement (typically rapid spin). Furthermore, beyond the avatar's vision field, the vision set takes into account the features of the game world, known by all players. For instance, the avatars that are in a player's vision range, but behind a wall do not appear in his vision set. This greatly reduces information available to players for map-hacks and sniffing.

Information Dissemination: The player receives infrequent (one per second in our implementation) dead reckoning messages (*guidance messages*) from players in its VS, which contain information such as the current velocity, future position predictions, and AI guidance instructions that enable the player to simulate the avatar's near-future actions. We have described how accuracy of such predictions can be greatly improved [16].

Interest Set (IS): it is composed of visible avatars that catch the player's attention the most (measured by a combination of proximity, aim and interaction recency). These are the avatars the player is most likely to interact with, therefore, requiring detailed information. Given the limited attention span of human players [5], the size of the IS can be fixed (e.g., 5). Only avatars in a player's vision set are considered as candidates, thus preventing the player to obtain frequent and accurate information about avatars he cannot see.

Information Dissemination: the player receives frequent (i.e., every 50 ms) state updates from players in the IS, including the avatars position, aim, ammunition, weapons, health, etc. Avatars in a player's interest set are automatically removed from its vision set to prevent him from receiving information about future actions he could exploit to cheat, as he is likely to interact with these avatars in a near-future.

Others: any player outside the VS and IS belongs to the *others* set. While players do not actually need information about the players outside their vision sets for rendering the game world, they still need a minimum amount of information to determine whether a given player does or does not enter their vision sets.

Information Dissemination: to allow players to decide on the type of subscription they need, they receive infrequent (i.e., typically every second) partial state updates containing only the position of the avatars, sufficient to determine the subscription type. This subscription type is assigned by default and thus it does not require explicit subscription.

B. Proxy architecture

Typical to most games, our solution assumes that the game runs in a discrete event-loop, dividing time into fixed-length frames. At any frame, a player has a single designated proxy (another player), responsible for managing subscriptions, forwarding messages, and for verifying the player's actions. As such, the proxy has tasks similar to a server.

Proxy assignment is done in a random, but verifiable way in order to prevent collusion from happening. To achieve this, Watchmen utilizes pseudo-random number generators: each player maintains a pseudo-random number generator for each player, including himself, initialized with the player's id and a common seed. This means each player can determine both its own proxy and the other players' proxies, in any given frame, without the need for communication. All players act as proxies for other players (unless performance or the security mechanism by design favor some nodes, see Section VI). Moreover, there is no need for explicit subscription by the players to their proxies as both parties are aware of their assignment to each other. Proxies are renewed every couple of seconds. At the time of this renewal, old and new proxies go through a hand-off mechanism that exchanges player state information for verification purposes.

Figure 3 shows how messages flow through the proxy architecture. Basically, all messages *sent* by the player are sent to his proxy: subscriptions, guidance messages, frequent

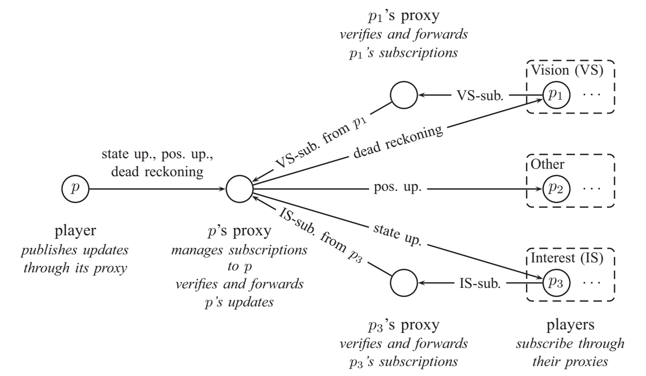


Figure 3. Proxies (1) manage incoming subscriptions and (2) verify and forward outgoing subscriptions and updates. Players p_1 and p_3 respectively IS-subscribe and VS-subscribe to p by sending a subscription message to p's proxy through their own proxies. By default, p_2 receives infrequent position updates.

state updates and infrequent position updates. A player's proxy relays subscriptions to the proxy of the target player to which the player wants to subscribe. That proxy has a list of all players that are interested in its player, and the player itself does not know who it interested in him: a subscription flows from the player to his proxy and from there to the proxy of the target player. In contrast, the proxy sends update messages directly to the players that have subscribed, no redirection through their proxy is needed. That is, an update message flows from the player to his proxy and then copies are sent to the players in the proxy's subscription list

IV. Security Aspects of the Proxy Architecture

While it is quite straightforward to see how the subscription model enables the prevention of unauthorized access, it is less clear how the proxy architecture contributes to cheat detection and prevention. In fact, our proxy architecture has been built from ground up with the goals of (1) preventing as many security flaws as possible, (2) provide sufficient means for verification of player's information and action, and (3) meeting performance requirements. Here we list some of the main architectural security advantages of the proxy architecture:

Verification Potential: The proxy is in a unique position to verify subscriptions and updates. It enables the *detection* of *flow* cheats such as escaping, look ahead, fast-rate, suppress-correct, replay, and blind opponents cheats, since proxies know exactly the required updates and the correct rate. Verification is discussed in more detail in the next section.

Consistency Cheat Avoidance: As a player only sends updates to the proxy who relays them, players cannot perform consistency cheats, i.e., send different updates to different players. If a player sends direct updates, it is detected.

Secured Subscriptions: Multi-resolution schemes are necessary for achieving the desired scalability. However, these approaches create an easy and yet very important opportunity for cheating: by sending subscriptions for high frequency updates, a cheating player is directly informed of interested players that are likely to target him. This information can be easily extracted, even without tampering with the game code, by using a simple network rate analysis tool during the game play. Watchmen's proxy scheme ensures that players are not informed about subscriptions to them.

Random, Verifiable, & Dynamic Proxies: In all distributed gaming architectures, part or all of the players' traffic goes through other players. This can be easily exploited by cheaters. In Watchmen, proxies are *random* meaning have no control over who they are a proxy for or who is their proxy. They are *verifiable*, meaning all players in the game can verify each other's proxy and automatically send to the correct proxy. And finally, they are *dynamic*, meaning the proxies are rearranged after a predetermined period of time (40 frames in our implementation). This is done by each player using the same random number generator to determine its own and other players new proxies. Thus, a cheating proxy can only disrupt a single *other* player's updates, only for a very limited period, with no direct benefit to his own avatar, and with high possibility of detection, therefore, greatly reducing his motives to do so. Furthermore, the dynamism of our approach makes any form of collusion very difficult.

Collusion Resistance & Verifiability: While most gaming architectures and even security mechanisms completely ignore collusion between players due to the difficulties in dealing with it, it is a present problem. In order to address this, Watchmen greatly reduces the information available to colluding cheaters. Proxy renewal is designed to limit the cheating and collusion opportunities available to cheating or malicious proxies. The proxy period is chosen long enough to be able to cross-check updates, but not long enough for colluding cheaters to cooperate in their actions.

Encryption & Signatures: To prevent proxies from tampering with the messages they forward – namely updates, subscriptions and handoff messages – Watchmen uses lightweight (i.e., $\sim$100 bits while state update messages are 700 bits on average) digital signatures [17], and each player verifies the digital signature of the messages it receives. This also prevents replaying and spoofing.

Handoff: is performed between a player's successive proxies to allow longer-term follow-up: before a player's proxy is renewed, it sends a summary of the player's state to the player's next proxy, i.e., its own successor. In addition, to limit the impact of player-proxy collusion, a proxy also embeds the summary it has received from its predecessor (follow up on two previous proxies).

V. Verifications, Reputation and Punishment

Not all cheating types can be prevented, therefore, most architectures rely on detection to deter players from cheating. For successful cheat detection we need to make sure: (1) enough information is available to perform efficient verification, (2) verification methods are successful in finding cheaters, and (3) detection results are useful for a blame/punishment system.

A. *Verification*

Each player can perform verifications of each other player. The types of verifications and their accuracy depend on whether he is the other player's proxy and/or whether he has the other player in his IS or VS.

The verifications done by the players are as follows:

Subscriptions: The proxy of a player p can verify whether a subscription of p to player q is justified based on p's position, aim, and movement history. This information is available at the proxy. A VS subscription is only valid if q is in p's vision cone. For incorrect VS subscriptions, the distance between the q and p's vision cone is used as a metric of the deviation. For IS-subscriptions, a proxy computes *interest* with sufficient accuracy based on the attention metric.

Position Changes: of a player p can be verified depending on the information available by the verifying player. As the proxy and players that have p in their IS receive the exact frequent position updates, they can easily compare successive updates and control whether the movements follow game physics (e.g., gravity, limited velocity, angular speed, permitted position). Proxies and players that have p in their VS range can *a posteriori* verify if the actual movements of a player are consistent with the predictions included in dead reckoning messages. We use the area between the simulated and the actual trajectory of the avatar as a metric of the deviation. Players that do not have p in their IS or VS can also do verifications as they receive infrequent update messages, but the accuracy is obviously reduced.

Actions: Similar to position changes, the proxy and the players that have p in their IS are able to verify that p's actions follow game rules (e.g., decreasing health after a fall) comparing successive state updates.

Dissemination Frequency: Proxies can control whether a player sends timely updates and other players verify that proxies forward them.

Interactions: such as hit and kill-claims are verified by proxies and by players acting as witnesses. The verification consists of checking that, e.g., a rocket was effectively fired and the distance between the position of the rocket and that of the target is used as a metric of the deviation.

Players are in charge of the short-lived objects they create, in addition to their avatars. Hence, such objects are checked by proxies and other players as well.

For efficiency reasons, we perform *sanity checks* [18] to detect cheating. However, *action repetition* checks (e.g., tamper-resistant logging mechanisms [19]) that would provide more accuracy but incur higher costs are also possible.

As most of these sanity checks are not exact (some information may be slightly outdated or be an estimation/prediction), each action is rated from 10 to 1 with regards to cheating probability (10 most likely cheating, 1 most likely normal). To determine the cheating rating, the verifying player checks whether the observed behavior (movement, dissemination frequency, subscription, etc.) falls within the expected behavior. If yes, the cheating rating is set to one. Otherwise, the higher the deviation from a expected behavior, the higher the cheating score for the action. What it means for an action to fall within the acceptable range depends on the action types. A simple example is: when a player receives two position updates, it can calculate the speed and determine the movement to be acceptable if the speed is lower than the maximum allowed velocity. As another example, when comparing previous dead reckoning messages with a current position update, the action is acceptable if the area a between the predicted trajectory and the actual trajectory of an avatar is smaller than the average value $\bar{a}$ observed for honest players plus a certain tolerance (typically the observed standard deviation σ_a to keep the false positive rate acceptable). That is $(a - (\bar{a} + \sigma_a)) < 0$ indicates a valid dead reckoning message–position update pair and everything above is suspected.

These ratings are further modulated by a *confidence* factor, taking into account the confidence of the player in his rating. Confidence depends on the accuracy of the information available at the player: proxies are assigned high confidence c_P, players that have the concerned avatar in their IS or VS have medium high c_{IS} and medium low confidence c_{VS} respectively, and other players have a low confidence c_O ($c_P < c_{\text{IS}} < c_{\text{VS}} < c_O$). In addition, it takes into account the staleness of updates: discrepancy of a new update with a very old guidance message is assigned a very low confidence.

B. *Reputation & Punishment*

Because the detection system has false positives (due to e.g., message loss), a single detection of cheating does not result in banning of players. Instead, each player tags the interactions he has with other players as successful (if no cheat was detected) or as failed, and this information is fed to a reputation system. Such information can be collected by either (1) a centralized game lobby that manages access and logins and can thus ban the players or (2) a distributed reputation system. In its simplest form, a reputation system decides to ban a node, if the proportion of acceptable interactions of a player drops below a given threshold. This threshold is set, based on the success and false positive rates of the detection system. More elaborate reputation systems (e.g., [20]) incorporate the notions of confidence and credibility to modulate the nodes' reports, prevent bad mouthing, and deal with collusion by analyzing relationships between nodes (e.g., the game's social network), resulting in an improved robustness. The Watchmen detection algorithm can be plugged into any reputation system. The design of such systems is out of the scope of this paper.

VI. Performance Characteristics & Comparison

The performance of the architecture is of utmost importance. Measuring Quality of Experience (QoE) [21], quantitatively, has always been hard. However, factors that directly impact the user experience have been well documented, specifically in a well-studied game such as Quake III (and similar games) [9]. The main factors are *latency* and *bandwidth requirements* which in turn, combined with network *loss*, translate to loss of *timely* updates. We will show our architecture meets and exceeds these requirements and compares favorably with similar distributed architectures.

Latency: Most distributed architectures proposed for games rely on multicast trees [22] or routing hubs [4] who need several hops to get a message to the destination. The best case scenario is represented in a direct subscription systems [5] that relies on forwarding pools. These forwarders relay traffic for nodes with worse connectivity, resulting in two hops (from publisher to forwarder to subscriber). This is in essence similar to our model in which traffic is forwarded by players' proxies (we'll discuss low bandwidth support further below). Furthermore, we use the following optimizations to even further improve latency:

(1) In each frame players calculate their subscriptions for the coming frame and send the subscriptions ahead of time. In our experience [16], given small differences between frames, using current angular and physical momentum, subscriptions can be calculated. (2) Subscriber retention, in which subscriptions are maintained for several frames, eliminates subscription latency after the initial subscription. To decide on the retention period, one must calculate the average change frequency in subscriptions. In our experiments, nearly 50% of the players in the IS change after 4 frames, less than 10% last more than 30 frames. While this value can be slightly different for different maps, we found it to be fairly accurate for most gaming sessions. This value can be used in a timeout mechanism for subscriptions: subscriptions are kept for a predetermined number of frames. Only new subscriptions are sent out explicitly. In addition, note that it normally ($\sim 83\%$ in our analysis) takes at least one or two frames to become the center of attention after entering the IS. (3) In extreme cases, one can relax the first hop requirement, if bandwidth allows it, and remove forwarding proxy requirement at the cost of lower security. Note that the subscriptions will remain anonymous as they are handled by target's proxy (the receiver's proxy also requires concurrent but less verifiable copies to be sent to it).

Upload capacity & Fairness: Not all peers in a distributed system may have similar bandwidth capacities available to them. This is usually solved by using forwarding through more powerful nodes. However, if only low bandwidth players use forwarding, they will be at a disadvantage as they send and receive updates later than others. In our system all players' traffic is processed through proxies, therefore, it is fair to the players. In addition, the selection process can be refined, if necessary, to take into account resource heterogeneity. This means that using the same verifiable random generator players' with low resources are removed from the proxy pool and more powerful can become proxies for more than one player. In such a case a player with low resources needs to send a single copy of updates to its proxy. The disadvantage, however, is that this will increase proxies access to information and should be avoided unless necessary. Similar to most current systems a feasibility test can be run at the beginning of gaming session to determine if players meet the minimum requirements. Changes in the available bandwidth can be advertised (infrequently) piggybacked on the updates sent.

Churn & NAT: Most architectures have to deal with churn. In our case, updates sent between players also act as a heartbeat mechanism that easily identifies the players that have been disconnected or left. These nodes are removed in the next round, through an agreement protocol, from the proxy pool. For NAT support, Internet Gateway Device Protocol (using the `MiniUPnP` [23] library) is used to add translation rules at the router. If the protocol is not supported by the router (or disabled), NAT traversal through "hole punching" is employed using the STUN(T) library [24].

Hybrid architecture: One of the advantages of Watchmen is that if game servers exist they can be easily incorporated by providing the game lobby, extra bandwidth, and becoming the proxy for some or all players. Tasks can be delegated to players as soon as they are determined to be trustworthy.

VII. Evaluation

No game research framework can realistically emulate commercial multi-player games, which take years and millions of dollars to develop. To best showcase the abilities of our architecture, we have therefore chosen to run experiments with the popular Quake III game. It is currently the best studied open-source game, and its requirements have been well documented, and therefore can be used to validate the experimental results. Our evaluations are based on the following: (1) Our enhanced Quake III (Written in C) can support up to a maximum of 48 players (instead of 16). Given inherent Quake III limitations, the game cannot be scaled to more players. We compute the different sets (IS, VS, Others) inside the game. In addition, a tracing module has been added to the game that records in a trace file all important game information, e.g., different sets, players' position, aim, weapons, ammo, health, and speed, as well as items location, item pickups, shootings, and killing of players. (2) A replay engine (in Python) has been built that can replay game traces and generate the same network traffic repeatedly and under different networking and proxy architectures to measure different aspects of the performance (e.g., latency). It helps with repeatable experiments by

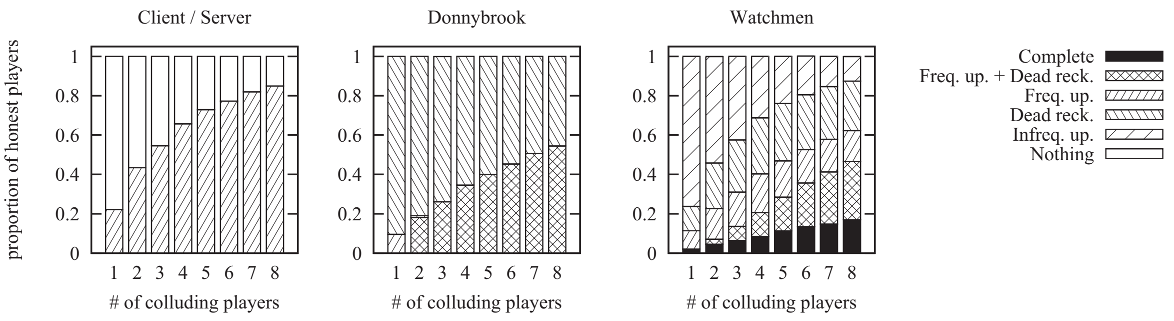


Figure 4. Information about players available to players (potentially cheaters). 48-player game in `q3dm17` map.

replaying the same game session over the network, exactly as Quake III would, using the collected traces (with various number of machines up to the maximum players) and is remotely deployable (e.g., over PlanetLab). Given availability of traces to all machines, each one can accurately measure which messages should have arrived and at what frame. (3) A trace-based simulation environment (in Python), that aims to measure the information exposure, scalability of the game, and the quality of verifications performed, using multiple traces and under different configuration settings.

The goal of our experiments is to show that Watchmen: (1) limits the effects of collusion and the information available to cheaters; (2) can detect cheaters using different types of verifications; (3) meets the latency and loss requirements already established for Quake III and similar games while keeping the scalability properties obtained by the distributed multi-resolution scheme[3]. We use the following for comparison: Donnybrook as a sample of a good multi-resolution system, where players receive frequent updates for players in their interest set and dead reckoning messages for others; and an optimal Client-Server case where players receive frequent updates for avatars in their PVS and nothing for the rest; and Watchmen as described in Section III.

Information Disclosure & Collusion: We evaluate how well Watchmen minimizes *information disclosure* through *messages*, even in the presence of collusion: Players can obtain several types of information about other players: complete information (i.e., proxies about the players they are in charge of), frequent state update (about avatars in the interest set), guidance (dead reckoning) messages (about avatars in the vision set), and infrequent position update. The first type is the most informative, the second and third complement each other, even though frequent updates are more detailed they are not directly *comparable* (as guidance includes extra player statistics and predictions for game rendering), and the fourth is the least informative. We measured the *joint* information obtained by a coalition of colluding cheaters about other players using a 48-player trace from a Quake III game in the `q3dm17` map. This is a worst case scenario as we assume all colluding players work together and any information available to one cheating player is *immediately* available to all colluding partners. We consider three infrastructures: Client-Server gives the minimum necessary information and thus serves as a baseline, Donnybrook as a sample multi resolution system (our implementation of interest sets according to Donnybrook, since the code was not available), and Watchmen.

[3]We do not evaluate distributed reputation schemes, since well established methods that cope with wrongful blames and collusion exist [20].

We compiled the results under the forms of stacked histograms, shown in Figure 4, as follows. Consider a simple example of a coalition of two cheaters for a game with eight players: player 1 with IS $\{4, 5\}$, VS $\{2, 6\}$ and other $\{3, 7, 8\}$, who acts as a proxy for player $\{3\}$; player 2 with IS $\{1, 6\}$, VS $\{7\}$ and other $\{3, 4, 5, 8\}$, who acts as a proxy for player $\{1\}$. The coalition therefore has: complete information about 1 honest player ($\{3\}$), frequent update **and** dead reckoning for 1 honest player ($\{6\}$), frequent update only for 2 honest players ($\{4, 5\}$), dead reckoning only for 1 honest player ($\{7\}$) and infrequent update for 1 honest player ($\{8\}$). It can be observed in Figure 4 that despite the information leakage caused by proxies, Watchmen significantly reduces the information disclosed when compared to Donnybrook. Our trace-driven simulations shows that Watchmen, a coalition of four cheaters has minimum information (i.e., infrequent position update only) for about 31% of the honest players and partial information (i.e., dead reckoning **or** frequent state update) for about 48% of them. In Donnybrook, the same coalition has dead reckoning information only for about 65% of the honest players and precise information (i.e., dead reckoning **and** frequent state update) for the rest. Since in Donnybrook players receive dead reckoning messages about all the players not in their interest set, the proportion of honest players about whom a coalition has frequent state updates alone is very low ($<$1%). In practice, Donnybrook makes use of forwarders (i.e., clients with high bandwidth multicasting updates for of clients with low bandwidth), which constitute a large and additional source of information exposure. Therefore, the results presented above are a **lower bound** of Donnybrook's information exposure. Note that these benefits are in addition to the subscription hiding gained by the architecture.

Effectiveness of Verifications: To showcase Watchmen's ability for verifications (1) we measure useful information about a player that is available to players (and in particular to proxies) that can be exploited to effectively implement most forms of verifications suggested in the literature. Furthermore, (2) we measure how often colluding cheaters are assigned to honest players, and (3) we measure effectiveness of several of these verifications in the form of sanity checks. Note that more complicated and rigorous action repetition verifications would only improve the results presented here.

In Watchmen, verifications rely on proxies and witnesses (i.e., players having sufficient information to assess the validity of actions). To evaluate the potential for effectiveness, we measure, for a given cheater, the average number of honest players that: act as proxy for him, have him in their IS, or have him in their VS. The results, shown in Figure 5, show great potential for verifications: even when a player colludes with 3 other cheaters (out of 48 players), he is assigned an honest proxy in 94% of the cases ($1-3/47$) and 10 players on average witness his actions (4 through frequent state updates and 6 through dead reckoning messages).

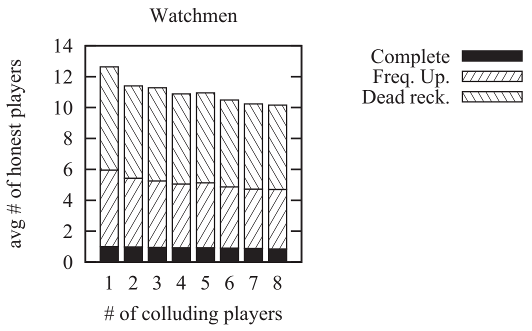


Figure 5. Levels of information about cheaters available to witnesses.

To further show the effectiveness of the information available, we set up an experiment where a cheater sends up to 10% invalid cheat messages. We measure the overall success ratio (high confidence detection by one of the honest players) of different verifications, where false positives (honest messages wrongly identified as cheats) are limited to a maximum of $\sim 5\%$. Note that this is regulated by the confidence factor of the players and how much deviation the update has from expected behavior and is very system and cheat specific. In practice, we manually and through experiments configured these values for different cheat types. However, these values can be properly obtained and configured by the reputation system used. The encouraging results are shown in Figure 6 for the following verifications performed by an honest player (other verifications would also be possible): **Position Updates:** Updates are verified to adhere to the game rules. For example, a player cannot move faster than allowed in the game. In our system, cheaters move randomly at 1.5–3 times the acceptable speed; **Kills:** One trivial opportunity to cheat in FPS is to unduly claim kills. We address this by verifying the type of weapon, the distance, the visibility, and how long the attacker had the target in his IS (Typically $4 \sim 10\%$ of the kills had their target in the IS for less than 2 out of 5 frames, depending on the map); **Guidance:** Guidance messages are compared against future frequent updates by the proxies as well as dead reckoning computed by proxies; **IS and VS Subscriptions:** Subscriptions are checked from the accurate information available to proxies.

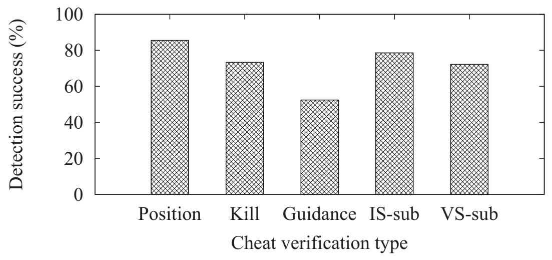


Figure 6. Success rates of the different verification mechanisms executed.

Responsiveness: To ensure that our architecture meets the official latency requirements of Quake III, we performed experiments on a LAN as well as preliminary tests on PlanetLab. Furthermore, we simulated latency in our networking module using latencies available from the King [25] and PeerWise [26] datasets, filtered using a Geo-IP location dataset that limits the locations of IP addresses to the United States (with mean latencies of 62 and 68 ms respectively). Our latencies are consistent with Halo's [12]. Message loss is simulated with a rate of 1%. Note that FPS games that receive updates within 150 ms latency with loss of under 10% deliver a good gameplay to users [10].

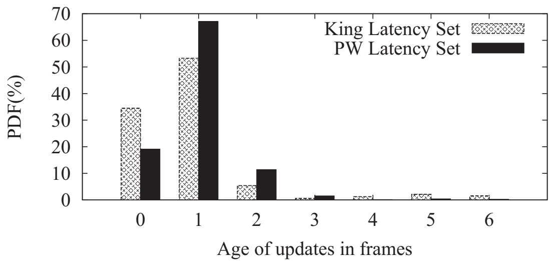


Figure 7. Distribution of the age of received updates (all three types) from the frame they should have been received.

The performance results for Watchmen are shown in Figure 7. Given that most P2P architectures use forwarding or multicast trees, Watchmen provides relatively low latencies and meets the requirements of FPS games. Quake tolerates up to 150 ms latency, therefore, only the messages that are 3 frames old or more of Figure 7 are counted as loss. One source of delay is when an avatar, neither in IS nor in VF, enters IS or VF. Indeed, the movement of the avatar should be displayed smoothly while only a possibly outdated position update is available. This phenomenon is however rather infrequent: in a frame, on average 88% of the players in IS were already in IS in the previous frame, 8.5% were

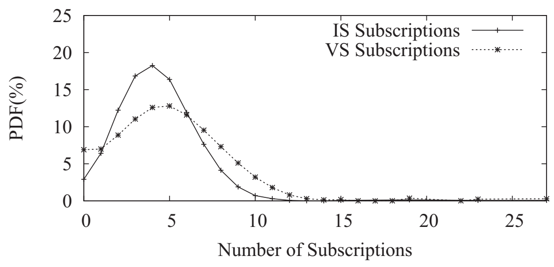


Figure 8. PDF of the number of IS and VS subscriptions.

in VS and only 3.5% would suffer from a slight delay. This can be dealt with by slightly widening the vision range (±60 degrees in our experimentation), thus increasing the responsiveness and the effectiveness of verifications, but also the amount of information disclosed.

Bandwidth: The upload bandwidth needed by our system is dependent on the IS and VS subscriber sets, i.e., on average how many players have subscribed to a given player. The distribution of these subscriptions is shown in Figure 8. This distribution does not significantly change as the number of players grows, thanks to the limited visibility and IS sizes. Even though our goal is not to further decrease bandwidth over existing multi-resolution schemes, Watchmen decreases the bandwidth usage for updates sent because of the employed information filtering. On the other hand, bandwidth is increased because of proxies, cross-verifications, and encryption. Upload bandwidth use is highlighted (per type of update sent) in Figure 9. Although the proxy architecture may incur some computational and bandwidth overhead, it does not hurt the scalability as, due to its decentralized nature, a player is in charge of only one other player on average, regardless of the total number of players. For example, Watchmen is able to scale up to 900 players with a 6 Mbps upload capacity.

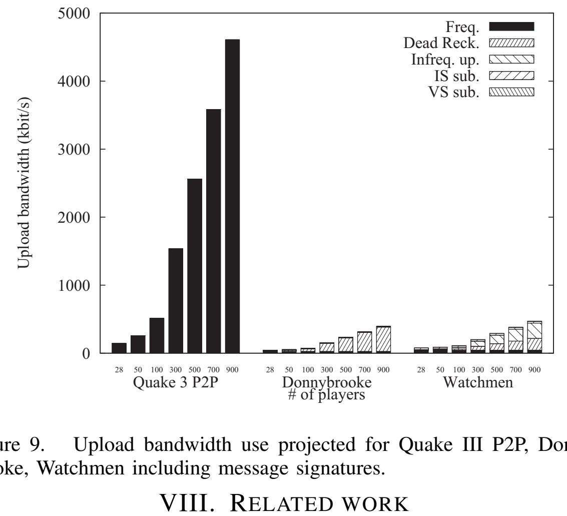


Figure 9. Upload bandwidth use projected for Quake III P2P, Donnybrooke, Watchmen including message signatures.

VIII. Related work

Distribution: Many distributed game architecture exist (e.g., [22]), however, few are designed for fast paced games like FPS. Use of DHTs [4] or multicast trees is too slow for these games. In addition, network overlays such as trees cannot follow fast paced changes in the players interest. Closest to our work is Donnybrook [5], which we use for comparison, a multi-resolution scheme for update dissemination. All of these architectures ignore cheating, i.e., they do not provide any cheat prevention and detection mechanisms, and even introduce additional cheating possibilities because messages between players are routed through other player nodes and at different rates. Note that there exist complementary tools that can be used by most architectures to deal with some types of cheating, e.g., secured client side verifications such as PunkBuster [27] and Valve Anti Cheat (VAC) [28], prevent the players from tampering with the game code. Pseudo-random number generators have been successfully used in P2P systems [29].

Mutual verification: A range of cheat detection approaches for distributed games, including mutual verification and log auditing, are described in [30]. Following these guidelines, Watchmen uses mutual verification, but only short-term auditing for improved responsiveness. It also mitigates unnecessary information disclosure. RACS [31] relies on trusted referees who simulate the game world locally to assess the validity of the players' action. IRS [18] considers the case where a server delegates tasks, including sporadic audit of other players, to one or several players, namely, proxies. One of the few papers that consider *collusion* is [32], where a referee selection algorithm optimizing both responsiveness and the probability of player-referee collusion is proposed for the case where (untrusted) players can be referees.

AVM [19] runs software binaries inside a virtual machine and uses tamper-resistant logging which allows for exact replaying of events and detection of cheaters. Technically, AVM can be used on our platform as others for added security. However, AVMs impose a non-negligible overhead on the game and log recording (particularly on now ubiquitous multi-core platforms) and exchanging them for verifications is non-trivial. Also, such platforms require that messages eventually be delivered, which is impractical in games, and do not protect against some cheats such as rate analysis and collusion. Moreover, deployment of VMs on game consoles for current-generation games is currently not available.

Unnecessary information exposure: In [33], a set of basic secure primitives are proposed to build cheat-proof P2P trading card games. A cheat-proof peer-to-peer implementation of each primitive is proposed, using cryptography.

Based on the fact that knowing additional information impacts the way players behave (e.g., a player obtaining information about players behind walls is likely to stare at walls), a statistical cheat detection approach has been proposed [34] where uncommon patterns in players' behaviors (e.g., movement) are identified. This is also possible in Watchmen through verifications run by proxies to further improve performance. Another singular approach to cheat detection [35] is to deliberately delay messages and analyze players reaction thus detecting typical look-ahead cheats, i.e., waiting for other players to declare their actions before committing one's own to take informed decisions (e.g., avoiding a rocket fired by another avatar). Other protocols (e.g.,[7]) make use of commitment schemes to prevent timing and look-ahead attacks, at the price of responsiveness.

SpotCheck [36] limits the information disclosed to the client by having the clients submit view requests to obtain information about the area that is visible to the player every time they move. The server probabilistically checks the validity of the view request based on position and visibility information Moreover, the server runs basic checks (e.g., size of the requested area) for all requests.

IX. Acknowledgments

Amir Yahyavi was funded by NSERC Strategic Grant STPGP/350626-2007. Kévin Huguenin was partially funded by a scholarship from University of Rennes I and an explorateur grant from INRIA. Julien Gascon-Samson was partially funded by a Doctoral scholarship from FQRNT.

X. Conclusion

This paper proposed Watchmen, the first distributed protocol designed for FPS games that aims in detecting and preventing cheating and collusion. This is done through cross verifications and vision-based filtering facilitated by a dynamic proxy scheme. Experimentations on Quake III show a great reduction in the information exposed to cheaters even in the presence of collusion. Furthermore, it shows great potential for effective verification of player actions by other players. Watchmen is able to scale well and can meet well-established latency requirement for fast-paced games.

References

[1] "Quazal," http://www.quazal.com/net-z.htm, 2011.

[2] "uLobby," http://muchdifferent.com/?page=game-unitypark-products-ulobby, 2012.

[3] M. Varvello, C. Diot, and E. Biersack, "P2P Second Life: Experimental validation using Kad," in *INFOCOM*, 2009.

[4] A. Bharambe, J. Pang, and S. Seshan, "Colyseus: A Distributed Architecture for Online Multiplayer Games," in *NSDI*, 2006.

[5] A. Bharambe, J. Douceur, J. Lorch, T. Moscibroda, J. Pang, S. Seshan, and X. Zhuang, "Donnybrook: Enabling Large-Scale, High-Speed, Peer-to-Peer Games," in *SIGCOMM*, 2008.

[6] L. Fan, P. Trinder, and H. Taylor, "Design Issues for Peer-to-Peer Massively Multiplayer Online Games," *IJAMC*, vol. 4, pp. 108–125, 2010.

[7] N. Baughman, M. Liberatore, and B. Levine, "Cheat-Proof Playout for Centralized and Peer-to-Peer Gaming," *IEEE/ACM ToN*, vol. 15, pp. 1–13, 2007.

[8] S. D. Webb and S. Soh, "Cheating in Networked Computer Games: A Review," in *DIMEA*, 2007.

[9] G. Armitage, "An Experimental Estimation of Latency Sensitivity in Multiplayer Quake 3," in *ICON*, 2003.

[10] T. Beigbeder, R. Coughlan, C. Lusher, J. Plunkett, E. Agu, and M. Claypool, "The Effects of Loss and Latency on User Performance in Unreal Tournament," in *NETGAMES*, 2004.

[11] G. Armitage, C. Javier, and S. Zander, "Post-Game Estimation of Game Client RTT and Hop Count Distributions," in *NETGAMES*, 2006.

[12] Y. Lee, S. Agarwal, C. Butcher, and J. Padhye, "Measurement and Estimation of Network QoS Among Peer Xbox 360 Game Players," in *PAM*, 2008.

[13] M. Claypool, "Network Characteristics for Server Selection in Online Games," in *MMCN*, 2008.

[14] A. Yahyavi and B. Kemme, "Peer-to-peer architectures for massively multiplayer online games: A survey," *ACM Computing Surveys*, 2013, to appear.

[15] "BigWorld technology," http://indie.bigworldtech.com.

[16] A. Yahyavi, K. Huguenin, and B. Kemme, "Interest Modeling in Games: The Case of Dead Reckoning," *MMSJ*, 2012.

[17] T. Eisenbarth, S. Kumar, C. Paar, A. Poschmann, and L. Uhsadel, "A Survey of Lightweight-Cryptography Implementations," *IEEE Design & Test*, vol. 24, pp. 522–533, 2007.

[18] J. Goodman and C. Verbrugge, "A Peer Auditing Scheme for Cheat Elimination in MMOGs," in *NETGAMES*, 2008.

[19] A. Haeberlen, P. Aditya, R. Rodrigues, and P. Druschel, "Accountable Virtual Machines," in *OSDI*, 2010.

[20] E. Ayday and F. Fekri, "Iterative Trust and Reputation Management Using Belief Propagation," *IEEE TDSC*, vol. 9, pp. 375–386, 2012.

[21] P. Chen and M. El Zarki, "Perceptual view inconsistency: an objective evaluation framework for online game quality of experience," in *NETGAMES*, 2011.

[22] B. Knutsson, H. Lu, W. Xu, and B. Hopkins, "Peer-to-Peer Support for Massively Multiplayer Games," in *INFOCOM*, 2004.

[23] "MiniUPnP," http://miniupnp.free.fr.

[24] J. Rosenberg, R. Mahy, P. Matthews, and D. Wing, "Session Traversal Utilities for NATs (STUN)," IETF, Tech. Rep. RFC 5389, 2008.

[25] K. Gummadi, S. Saroiu, and S. Gribble, "King: Estimating Latency Between Arbitrary Internet end Hosts," in *IMW*, 2002.

[26] C. Lumezanu, R. Baden, N. Spring, and B. Bhattacharjee, "Triangle Inequality Variations in the Internet," in *IMC*, 2009.

[27] Even Balance, "PunkBuster," http://www.evenbalance.com.

[28] Steam, "Valve Anti-Cheat System (VAC)," https://support.steampowered.com/kb_article.php?p_faqid=370, 2012.

[29] H. C. Li, A. Clement, E. L. Wong, J. Napper, I. Roy, L. Alvisi, and M. Dahlin, "BAR Gossip," in *OSDI*, 2006.

[30] P. Kabus, W. Terpstra, M. Cilia, and A. Buchmann, "Addressing Cheating in Distributed MMOGs," in *NETGAMES*, 2005.

[31] S. Webb, S. Soh, and W. Lau, "RACS: A Referee Anti-Cheat Scheme for P2P Gaming," in *NOSSDAV*, 2007.

[32] S. Webb, S. Soh, and J. Trahan, "Secure Referee Selection for Fair and Responsive Peer-to-Peer Gaming," *Simulation*, vol. 85, pp. 608–618, 2009.

[33] D. Pittman and C. GauthierDickey, "Cheat-Proof Peer-to-Peer Trading Card Games," in *NETGAMES*, 2011.

[34] P. Laurens, R. Paige, P. Brooke, and H. Chivers, "A Novel Approach to the Detection of Cheating in Multiplayer Online Games," in *ICECCS*, 2007.

[35] S. Ferretti and M. Roccetti, "AC/DC: An Algorithm for Cheating Detection by Cheating," in *NOSSDAV*, 2006.

[36] S. Moffat, A. Dua, and W.-C. Feng, "SpotCheck: An Efficient Defense Against Information Exposure Cheats," in *NETGAMES*, 2011.