Enforcing Game Rules in Untrusted P2P-based MMVEs

Sebastian Schuster
Department of Distributed Systems
University Duisburg-Essen
Bismarckstrasse 90
47057 Duisburg, Germany
sebastian.schuster@uni-due.de

Torben Weis
Department of Distributed Systems
University Duisburg-Essen
Bismarckstrasse 90
47057 Duisburg, Germany
torben.weis@uni-due.de

ABSTRACT

Cheating is a major obstacle on the way to realize MMVEs based on peer-to-peer networks of untrusted nodes. Current approaches either rely on trusted nodes or use a pessimistic distributed simulation, with negative effect on the realtime responsiveness of the virtual world. We propose to use an optimistic simulation maintaining the realtime immersion of players. We show how different kinds of cheats can be prevented or detected using this approach based on a broadcast network. Most importantly, tampering with the world state by malicious players is prevented. Finally, we show how to make this approach scalable by using a representative subset of nodes to decide on the validity of player actions.

Categories and Subject Descriptors

C.2.4 [**Distributed Systems**]: Distributed applications; I.6.8 [**Types of Simulation**]: Gaming

Keywords

Security, MMVE, cheating, peer-to-peer architectures

1. INTRODUCTION

Massively Multiplayer Online Games (MMOGs) like Blizzard Entertainment's World of Warcraft have become tremendously successful, attracting millions of players worldwide. MMORPGs are the predominant type of Massively Multiuser Virtual Environments (MMVEs). The once niche market has grown substantially. Consequently, competitors try to enter the market and a lot of new MMOGs are being developed. All of these MMOGs are based on client/server technology. The players' game clients send the player actions to a central server, the server updates the world state and sends the state updates back to all interested clients. This scheme is quite simple, the technology is mature, and the game provider has full control over the virtual world. On the downside, buying and operating powerful game servers is expensive. Furthermore, the lack of scalability makes provisioning server capacity difficult. One server can only support up to a few thousand players. Depending on the popularity of the game, server capacity is easily under- or overprovisioned. If the capacity is insufficient, players will be unhappy with the bad game experience at peak times because of the overloaded servers. On the other hand, servers running idle waste resources and money.

Therefore, realizing dezentralized distributed virtual environments based on peer-to-peer (P2P) technology has received a lot of attention in the last years. In an ideal peer-to-peer system, every peer devotes at least as much resources to the system as it consumes, making peer-to-peer systems indefinitely scalable without any server cost for the provider. However, the bigger vulnerability against cheating is still an open problem, hindering the realization of P2P-based MMOGs.

An MMOG is a very competitive environment. A player takes over the role of a hero in a fantasy world. He controls his player character or avatar in the world to fight monsters. He wants to advance his avatar, get better equipment, to defeat the most challenging monsters or other powerful players. Becoming more powerful requires a lot of time and effort. Consequently, there will be some players trying to break the game rules – to cheat. Honest players are very concerned about the fairness of the game world. If cheating is possible, players will quit the game.

Sending all player actions to a central server checking them and storing the whole world state makes enforcing game rules straightforward in client/server-based MMOGs. In decentralized P2P-based MMOGs, the world state is distributed among the participating peers and checking actions for game rule compliance must be done in a decentralized fashion. Most importantly, we cannot trust the peers to run the unmodified code and to not tamper with the stored state. We need a mechanism to compensate for the lack of trust by employing additional redundancy to store the game state and check the game rules.

In this paper, we present our approach to enforce game rules in a P2P-based MMOG. First, we describe our system model and recapture the requirements of P2P-based MMOGs. We also describe the types of cheating that must be prevented. Afterwards, section 3 presents our approach to cheat-proof P2P-based MMOGs using an optimistic distributed event-based realtime simulation and broadcasting. We also show how current MMOG rules can be mapped onto this approach. In section 4, we modify the broadcast-based approach to use a scalable AOI-cast. Section 5 compares our approach to related work, while section 6 concludes.

DISIO 2011, March 21, Barcelona, Spain

DOI 10.4108/icst.simutools.2011.245550

2. FOUNDATIONS

2.1 System Model

In our system model, players use a game program installed on their Internet-connected PCs to join the virtual world. We assume player and thus node numbers to be in the range of thousands. Every node has more computational power and storage space than necessary to simulate the local view of the virtual world. We can use these spare resources to perform additional redundant calculation and storing of game data. Direct communication between nodes is reliable and message ordering is preserved. This is either achieved by TCP or a reliable UDP protocol.

The nodes have physical clocks we can use to obtain wall clock timestamps. The clocks of non-malicious nodes are always synchronized within tight bounds. Using NTP clock synchronization within 10 ms should be possible.

We assume the game provider operates a certification authority issuing user certificates to control access to the game. We can use these certificates to identify players. Since a node is under complete control of its owner, we cannot expect it to run the unmodified game program. A malicious player might have its node show byzantene behavior, reading and writing to the local storage and exchanging any kind of message. Furthermore, he might have multiple identities and nodes for coordinated misbehavior. However, we assume there is an honest majority of players. After all, cheating only makes sense when there are enough honest players to cheat. Without trusted nodes, if there is a malicious cooperating majority, it can simply change the game rules in any way.

The virtual world consists of a set of objects. The state of all these objects makes up the virtual world state. There are static objects, never changing their state. Static objects typically make up the biggest part of the world, representing the ground, buildings, furniture, trees, and so on. There are also dynamic objects changing their state over time according to the game rules. This includes the position of an object, if it is mobile. Characters are a typical example for mobile objects.

Simulating the virtual world is done by updating the virtual world state several times per second in a simulation update loop. In each update, the old world state is inspected and based on the time passed since the last update, the new state of all dynamic objects is calculated. For example the position of a moving entity is calculated by adding its current movement direction, multiplied with its movement speed and the time passed since the last update to its old position. In the same way, the state of AI-controlled characters or the state of actions performed by a player character are pushed forward.

2.2 Requirements and Types of Cheating

We have already provided a detailed requirement analysis of P2P-based massively multiuser online gaming [14]. Therefore, we will only briefly recapture those related to our anti-cheating approach.

First of all, virtual worlds must allow fast-paced realtime interaction. Players expect nearly instantaneous reactions of their avatars and its surroundings to player input. They also expect the virtual world to evolve according to the game rules. Players expect the world state to be determined by the performed actions they perceived. Players will tolerate sudden unexpected changes of the world state only to a certain extent and only if they do not happen too often.

The virtual world should also be scalable, being able host thousands of players and adapt to changing player numbers between low and peak times.

Finally, players expect fairness. The game rules should be the same for all players and cheating should not be possible. This includes all players have to see the same world state at the same time, so the game world has to be consistent across nodes.

There are many different ways how a malicious player might try to get an unfair advantage over honest players. First and probably the most severe kind of cheat is world state tampering. A cheater could try to alter its avatar's level and equipment, making it stronger or weakening other players. He could also modify its position, teleporting his avatar around or make it pass through walls. If a cheater could directly change the world state, he could basically do everything. This cheat must be prevented at all costs.

Second, a cheater could also try to break the game rules by performing actions that are not allowed at a certain time according to the game rules. Third, he could try to generate actions for honest player, forcing them to perform bad actions. Another goal of a cheater is to provide false information about the world to players, making the world inconsistent and leading the other players to take wrong actions. He could send out conflicting actions to different other players or he could also try to blind a player by not sending him any actions at all for a certain time.

There are also other kinds of cheats. Gaining more information about the world than allowed by the game rules is an example. Timing cheats, where nodes delay messages or modify timestamps of messages are another severe problem. These types are not yet covered by our approach.

3. BROADCAST-BASED SIMULATION

Our approach is based on the idea of Timewarp [13]. Timewarp runs a fully deterministic simulation replicated across all nodes. All events are timestamped using a physical clock and broadcasted to all nodes in the system. Consistency is ensured by executing all events at the same virtual wall clock time on all nodes. Local lag is used to hide network latency. Our main contribution here is to show how game rules of current MMOGs can be implemented using Timewarp's event-based approach. Using this approach, we can protect the world state from being tampered with by malicious nodes.

The basic idea is to exchange player actions instead of state between nodes. Whenever a player executes some action, all nodes check whether executing this action is allowed. Then, this event will be sent to all other nodes. Assuming all nodes have the same state before this event, after receiving and processing the event they will end up in the same new state. If a malicious player modifies his client to execute an action not allowed by the rules and sends this event to other nodes, they will not execute it if their game rules are correctly implemented.

To make a distributed realtime simulation completely consistent across all nodes, they would have to execute all events at exactly the same wall clock time. On a PC without realtime operating system support, it is not possible to force the game update loop to perform updates at exact points in time synchronized across all nodes. Therefore, a node uses

a time-ordered event queue to schedule timestamped events for processing at their designated execution time. When the game update runs, the world state is not immediately updated to the current wall clock time. Instead, all scheduled events with a timestamp smaller than the current time are processed. Starting with the oldest event in the queue, the node updates the world state and advances the virtual time to the time of the event, processes the event, removes it from the queue, and continues with the next event. Using this mechanism we can make sure that the same event is executed at the same virtual time resulting in an identical updated state. With closely synchronized clocks, the events will also be visible to the user at the same time.

However, we have to make sure all events are processed at all nodes in time. While we assumed reliable communication, actually transferring the message might take an arbitrary amount of time. Events may arrive when the time already advanced to a time after the event timestamp. There are two ways to circumvent this problem: the pessimistic way is to advance the virtual world time only when all nodes have acknowledged they are not going to generate an event laying before the new time [5]. This makes the node with the highest communication delay slow down and dictate the speed at which the simulation advances, violating our realtime requirement. The optimistic approach as implemented by Timewarp always advances the virtual time with wall clock time. Every node stores the game state in certain intervals and logs all events. When a late event comes in, the simulation is rolled back to the latest preceding state and all events including the late event are reprocessed. This approach is better suited to guarantee a realtime flow of action under considerable network delay. On the other hand, rolling back the simulation means the state of the world from the time of the late event until the current time was invalid and will suddenly change from one update to the next. This inconsistency might be very irritating for the player. However, using local lag [13] by adding a certain offset to the event timestamp after the player generated it, delays the event processing time. Thus, the player will not get immediate response after triggering an event. A local lag in the area of 100-160 ms was shown to be acceptable to players maintaining the realtime immersion [13]. If network delay and clock deviation are smaller than the local lag, the event will not cause a rollback at the receiver.

In the following, we will explain how typical MMOG rules can be realized using the event-based realtime simulation.

3.1 Player Movement

Instead of exchanging player positions in fixed intervals applying dead reckoning [3] for smoothing and extrapolation, movement and rotation vectors model the movement state of an entity. Whenever a world update occurs, the position and direction of a player character is updated based on the old position and direction, the current movement and rotation vector, and the time passed since the last update. Since it is quite unlikely that world updates run at the same time on different nodes, player character positions will not be consistent most of the time. However, they will always be consistent when an event is being processed and no rollback occurs, as the world and thus all entity positions are first updated to the time of the event. Consequently, when the player presses or releases a key to start or stop moving or rotating, he generates a movement change event. This event will be timestamped using the local lag and sent to all other nodes. When processing it, the position of the player character at the event time is calculated and then the movement or direction vectors are updated. No matter how long the sequence of player movements and how many intermediate updates are being calculated, player character positions will always be consistent when any event occurs as the world state is always updated to the time of the event. They will also be consistent when player character stopped moving, as the position does not change anymore.

However, usually players are not free to move wherever they want. They will collide with static objects like walls and stop. In some virtual worlds they will also collide with dynamic or moving objects. Usually, a geometrically simpler bounding sphere around every collidable object is used to calculate collisions. In each game update, the bounding sphere is checked for collisions with other bounding spheres. If it collides, the movement is stopped. As object positions have to be consistent, this type of collision detection cannot run within the regular world update. Instead, the collision detection has to be scheduled to run at certain times e.g. in fixed intervals. Depending on the maximum speed of moving objects and the size of bounding spheres, a minimum frequency for collision detection events is necessary. Otherwise it might be possible to pass through walls if the distance travelled between two collision detections events is big enough to move from one side of a wall to the other.

3.2 Other Player Actions

Besides movement, player characters can perform actions like attacking an enemy or casting spells. There are game rules defining the conditions under which these actions may be executed and the times certain effects occur. For example casting a spell takes a certain time. When the player finishes casting the spell, its cooldown is triggered so the same spell may not be cast again for a certain amount of time. Casting a spell also costs resources which will be subtracted from the player character's resource pool when casting finished. After that, the target is hit by the spell and a certain amount of damage based on the power of the caster and the target is caused.

To realize this kind of timed action, we generate a start event for this action, apply local lag and send it to all other nodes. When the start event is processed, the preconditions for this action are checked. Preconditions might be that the cooldown for that action must not be active, the target must be in range and enough resources must be available. If the preconditions are met, a new event is scheduled in the event queue marking the completion of the cast time. Continuously running game updates can be used to update the state of a progress bar in the user interface. The world state change only occurs when the completion event is processed. This event will subtract the resource cost of the spell from the resources of the player character. It will also activate the cooldown for the spell. The completion event will also schedule two new events. First, the event to hit the target and cause damage will be scheduled for a future time. Second, the event to reset the cooldown will be scheduled. Both events will be processed subsequently. It is important to notice that all events generated after starting the action are internal events, they are not sent to other nodes. However, they will be executed at the same time since the start events are executed at the same time. The scheduled times

and the effects of the events are all part of the game rules. Executing these events will generate the same result on all honest nodes, without any further communication.

3.3 Non-Player Entities

Using the event mechanism, the behavior of AI-controlled game entities (non-player character, NPC) can be synchronized without any communication costs. Normally, the AI state of an NPC is updated as part of the game update loop. The AI inspects the state of other entities in the world and decides where to move or what actions to perform. By using events to schedule this process for predetermined times e.g. fixed intervals, the AI on all nodes will inspect the same world state at the same virtual time taking the same actions. It can decide to change its moving and rotation vectors to follow a close target with the game update loop providing the smooth movement similar to player character movement. Often AIs implement some kind of random behavior like randomly wandering around. A deterministic pseudo random number generator with the same initial state can be used to obtain the same random number at each event on all nodes. It can also be used to have random results for events like hitting or missing a target with a spell.

When a player defeats an NPC and it dies, a new event will be scheduled to remove the corpse of the NPC after a certain time. After removing the corpse, an event will be generated to respawn this NPC after a certain time. Thus the game world keeps regenerating its entities just like it is done in current client/server MMOGs.

3.4 World State Initialization

Timewarp ensures the game state is eventually consistent among all honest nodes. However, nodes must have the same initial state so Timewarp can ensure it stays consistent. Nodes constantly join and leave the world. Thus we need a way for new nodes to obtain the current world state. As we will show, wall clock timestamps alone are not sufficient for this purpose. In the following, we will explain how a combination of physical clocks and vector clocks can be used to obtain the current world state.

After joining the underlying communication network, a joining node asks all other nodes for their current world state and starts listening for events and responses to its query. The other nodes will send a timestamped copy of their world state while continuing to send and process events. Since the requests might have arrived at different times at the other nodes, the world states arriving at the joining node will have different timestamps and the object state might be different. Using the events recorded after sending the state request, a joining node can update the received world states to the current time. All updated world states of honest nodes should be equal. Assuming the majority is honest, a node will pick the correct world state. However, the joining node cannot be sure the world state it received is actually valid. Its originator might have received a late event after sending the world state out and consequently performed a rollback. This world state cannot be used at the joining node. However, a node can detect this using vector clocks.

Every node counts its generated player events. Whenever an event is broadcasted, the event number is included. When an event is processed, its event number is used to update the vector clock timestamp of the world state. This vector clock consists of one component for each player character in the world. We already described how to apply sparse vector clocks to be used in MMVEs [18]. The vector clock timestamp of a world state can be used to find out whether an event is included in the world state. If the world state's vector component of the event originator is bigger or equal to the event number, the event is already included in the world state. If not, the event is not included. If the event is not included and the timestamp of the event is newer than the world state then the state is valid with regard to this event. The event can be used to update the world state to the event time. If the timestamp of the event is older than the world state time, this event should be included. However, the vector timestamp says it is not so the world state is invalid.

If the joining node receives an invalid world state from a node, it will request a new current world state from this node until it finally receives a valid world state. Note this might take a very long time if the latency to this node is bigger than the chosen local lag. This means every received event leads to a rollback at that node. Only when no events are happening for some time, this node is actually able to send out a valid state.

After receiving the world state from a node, the joining node might miss events to update this world state to the vector time of the newest world state it received. This could happen when an event happened before the node started joining the world, so it did not receive the event. At the same time, some other node has a very high latency towards the event originator and this event is still in transit. Thus, the joining node might receive a world state without this event from the high latency node without a chance to receive the event from its originator. Again, the joining node will have to request a new world state.

When a player wants to leave the game world, he can generate an event to turn his character invisible and deactivate him. The remaining nodes will continue to store the characters state as part of the world state. After successfully retrieving the initial world state, the player can generate a matching event to activate his character. Relying on an honest majority of nodes and at least one node in the game world, the correct world state is persistently stored even when nodes join and leave.

3.5 Dealing with Cheaters

By exchanging player actions only, we can make sure the world state only evolves according to the game rules on honest nodes. Thus, the first type of cheat, state tampering, is impossible.

The second type, trying to execute actions when not allowed is also prevented. As all honest nodes will finally obtain the same world state when they receive all events, they will also deny to execute actions not allowed by the game rules at a certain time. However, it may happen the world state on a node is rolled back. Thus, an honest node may temporarily execute an action that is not allowed according to the true world state. Until the late event arrives and the true world state is calculated, the honest player will see the cheating player perform actions that are not allowed. However, these actions will be undone later.

After detecting this case the honest node cannot conclude the other node is cheating because it should not have sent out this action in the first place. The other node could have initiated an action based on a world state that was rolled

back later. Thus, the other node cannot be punished for a cheating attempt.

The third type of cheat, generating actions in the name of other players can be prevented using signatures. All events are signed using the player ID assigned by the central certification authority. These signatures are checked at the receiver using the certificate of certification authority.

To detect when a player sends conflicting actions to different nodes, nodes have to compare their world states on a regular basis to see if they diverge. Using checksums will be useful to speed up this process. The world state used for comparison should not be the most recent one so it is unlikely it will be rolled back. When a difference is detected, the log of actions must be compared. If conflicting actions are found, one has to be picked as the true one. It must be sent to all nodes and if nodes processed the wrong action, they have to perform a rollback and delete the other action. Finally, the conflicting actions can be presented to the game provider as a proof and the cheater can be excluded from the game.

The fifth type of cheating, malicious nodes suppressing updates to honest nodes is the most difficult case. When receiving non-consecutive events according to the event number or when comparing its world state, the honest node will finally detect it is missing events. Using the vector timestamps of the world states from the other node, it can find out which events it is missing and request these events from other nodes. Thus, it is able to obtain the correct world state. However, until this time the node was successfully blinded by the malicious node. Furthermore, the honest node has no way of proving that the malicious node did not send her the message. All it can do is raise a claim that the malicious node suppressed events. However, without proof a malicious node could also raise false accusations. If a claim is raised, it is only certain one of the two nodes is malicious since the network is assumed to be reliable.

There are three ways to deal with this type of cheat. First, the number of times a node is present as the one claiming or the one claimed to be cheating can be counted. When a certain threshold relative to the total number of claims is reached, the central certification authority can finally decide to exclude a player for suppressing events or for raising false accusations. Second, after raising a claim, a node can ask the other nodes to forward all events from a node it accused of suppressing events. Thus, he will only be blinded for the very short time of additionally forwarding the message. Third, honest nodes can employ a tit-for-tat strategy to blind nodes blinding them. The first and the third approach should motivate potential cheaters not to try suppressing events, while the second approach alleviates the problem of being blinded by reducing the blind duration. All of the three approaches can be combined.

4. SCALABLE SIMULATION

We have shown how a fully replicated optimistic simulation like Timewarp can be used to realize current MMOG game rules in a P2P broadcast network. We described how it is able to deal with five types of cheating.

However, the sketched approach has a lot of drawbacks. The event-based realtime simulation is computationally expensive. The time it takes to compute a world state update mainly depends on the number of dynamic objects in the world. Without using events, typical 3D game engines like Microsofts's XNA calculate around 60 updates per second as default. When using events, each of these updates will trigger an additional world state update for every event scheduled for a time since the last update. However, not every type of event needs the whole world state updated so optimizations are possible.

The main reason this approach cannot be used in practice is its lack of scalability. It means every node has to store the whole world state and simulate the whole world. While we assumed computational power and storage space is available, we expect MMVE worlds to be quite large, so this will not be feasible. However, the most severe problem lies in the network and communication structure.

Maintaining a full mesh network and broadcasting all events to all nodes will overload the network even for very small node numbers. We have developed a prototypical implementation, using a graphical 3D game client on top of our Peers@Play networking framework providing a full mesh overlay network with broadcast capability. We were able to run a virtual world with a very small number of nodes.

As we increased node numbers and players generated a lot of events, the network was overloaded soon and rollbacks kicked in, just as we expected. However, we could see that the increased update duration was not an issue for the processing power of modern PCs, even when a lot of events had to be processed. Thus, scalability is the remaining major concern we have to deal with.

In the following section, we will describe how we want to adapt our approach of exchanging player events only to obtain a scalable simulation.

4.1 Scalable Event Exchange

It is well-known that the key to achieve scalability is the fact that a node only has to know the world state in the local area of the player character. This area of interest (AOI) resembles the range-limited perception capabilities of human beings. This means a node only has to store the world state within the AOI and the world state can be partitioned and distributed among all nodes. Furthermore, each node is only interested in events happening within its AOI.

A locality-aware overlay network mapping the positions of player characters in the virtual world to node positions in the network can realize an AOI-cast [10]. It allows sending an event to all nodes whose AOI overlaps the event's position in the virtual world. Thus, all nodes interested in the event can update their part of the world state. The number of exchanged messages does not depend on the total number of nodes as is the case when using broadcasts. It depends on player density. If players are distributed all over the virtual world, this approach provides the desired scalability.

However, if we combine AOI-cast and world state partitioning with our approach to exchange player events to enforce game rules, two challenges arise. First, nodes only store a part of the world state. When the node moves into a different area, it must obtain the current world state in that area so it has to exchange state information. We also have to make sure the world state is stored for areas nobody is currently in. Second, if only a subset of nodes maintains the world state in a certain area and this subset depends on the nodes' positions, it is easy for a malicious player to move all player characters under his control to a certain area and modify the world state there.

Since sending an event to all nodes does not scale and

sending an event to close nodes only can be easily exploited, we propose to additionally send an event to a representative subset of all nodes. The size of this subset is some predetermined constant k. The members of this subset have to be picked based on properties an attacker can hardly influence. The certified player ID assigned by the game provider cannot be influenced by the attacker. If it is bound to the player's real life identification or if it is at least expensive to get an ID, it is very hard for a malicious player to influence his position in the ID space. We have already developed an approach to place data replicates equally spaced in a distributed hash table [15]. We use a similar approach to pick the nodes an event must be sent to additionally.

If a player has an ID id and the size of the ID space is n, then the subset of nodes this player has to send any event to will be the nodes closest to the IDs $id + i * n/k$ for $i = 1, ..., n-1$. This *k-set* of nodes has to store the events from a player and every node knows where to find the stored events of that player. When inconsistencies between world states on different nodes are detected, nodes can find out which events they are missing using the vector timestamps of the world states. Then, they can request the missing events from the *k-set* to calculate the correct world state within their AOI.

An event is only regarded as valid if a majority of the *k-set* agrees on the event. Otherwise a cheater only needs to control one additional node in the *k-set* of its node to create any allowed sequence of events for its node a long time later, which would be a timing cheat. Choosing k is a tradeoff between the number of exchanged messages and the probability the majority of honest nodes in the network is preserved in the *k-set*. The bigger k, the less likely an attacker can control nodes at more than $k/2$ specific positions in the ID space.

The overhead of sending events to k additional nodes is significant. However, it only depends on the constant k and not on the the total number of nodes as is the case when broadcasting events. Thus, the communication scheme is scalable. In addition to providing AOI-cast, the underlying network has to support sending of messages based on IDs, so events can be sent to the k nodes independent of their positions in the world. It also has to make sure the stored events are transferred to neighboring nodes when nodes leave gracefully and use additional replication in the neighborhood to increase reliability.

4.2 Dealing with Cheaters

In the following, we will show that our scalable approach is able to deal with the same cheats as the broadcast-based approach with a certain probability.

Similar to the broadcast-based approach, nodes have to compare their world states on a regular basis. When a node detects a different state of an object in its local world state compared to the object state on another node, both nodes have to decide on the correct state. Then, every node has to show how its world state came to be by providing the sequence of events leading to this world state based on a common initial world state. The common initial world state will be provided by a reliable and secure storage as described in section 4.3. Even if both nodes are honest, it can always happen that a node missed some events because not every event is sent to every node. However, comparing the vector timestamps of their world states, nodes can find out whose events they are missing and they can request them from the *k-set* of the event creator. Assuming the initial state is correct and the majority of the *k-set* is honest, the world state evolves according to the game rules only and it cannot be tampered with by cheaters. Similarly to the broadcast-based approach, actions not allowed can only be executed if the world state is temporarily inconsistent. Finally, honest nodes will always obtain the true world state and they will deny actions not allowed. Thus, cheat types one and two are not possible.

While processing events retrieved from the *k-set*, conflicting actions can be detected. Again, nodes must decide which event is the true event. If the majority of the *k-set* of the conflicting events' originator agrees on an event, this event is chosen. Nodes can obtain this event whenever they detect their state is not consistent with another node's state. If there is no majority for an event within the *k-set*, one event is chosen according to some predetermined criteria and the other events will be deleted. At the same time, the event's originator can be excluded by the game provider, providing the conflicting events as proof. Thus, cheat type four can also be detected and punished.

Cheat type five, suppressing events, is handled similarly to the broadcast-based approach. When an honest node detects it is missing events it should have received, it raises a claim and asks the *k-set* of the event's originator for the events. Since the *k-set* has to agree on an event for it to be valid, a node suppressing events in its area will always send these events to the *k-set*. Thus, the honest node will be able to retrieve the suppressed event asking the *k-set*. It can also request the *k-set* to additionally forward all events from the cheating node, receiving suppressed updates with a one hop delay.

A cheating node could also raise false accusations about another node suppressing events, which this node would of course deny. It cannot be decided who is right because not sending a message cannot be proven. Again, counting the times a node is involved in claims of suppressing events might finally be used to identify a cheater and exclude him from the game. Honest nodes using tit-for-tat by also suppressing events to the cheating node might further encourage nodes to not suppress events.

As events of a node are only stored in a subset of nodes, a cheater might also try to suppress events of honest nodes. However, to succeed a cheater would need a majority in the *k-set*, which is hard to achieve.

4.3 Reliable Secure Storage

Nodes must obtain the correct initial world state so they can calculate the correct current state. Furthermore, since nodes only store the current world state within their AOI, there must be a way to persistently store the world state for areas without any players. Additionally, we cannot expect the nodes to store all events they ever generated, ever received from other nodes in their AOI or from nodes whose *k-set* they are in. Over the lifetime of the virtual world this would require too much storage space and it would make the calculation of the correct state based on a common initial state too expensive. Therefore, we propose to use a reliable and secure storage to save the current world state at certain points in time persistently. Thus, nodes do not have to store events with timestamps before that time. We want to separate the concern of scalable realtime cheat-proof event

delivery from the concern of reliably storing the game state while protecting it from tampering by malicious players. As the persistent storing does not have to happen in realtime, we can pick a different design especially focusing on reliability and security.

When a player joins the virtual world, this storage is used to first obtain the state of the player character. His position is used to join the locality-aware overlay at the correct position. Then, the last stored state of the player character's AOI is loaded. This state is then used as the initial state to calculate the current state of the AOI based on the current state of the other nodes in the area, similar to the broadcast-based approach 3.4.

When players move around, the number of players storing a copy of a dynamic object might fall until the point where nobody holds a copy. Before that happens, the latest state of that object has to be stored persistently. The object state should also be stored in fixed intervals to lower the chances of world state being lost or malicious players being able to provoke a state loss, e.g. by letting all of his nodes in an area leave simultaneously. When a player enters an area without other nodes, it has to request the current state of this area from the secure storage.

The basic design of this storage is based on our previous work, using equally spaced replication with a replication factor k to realize a distributed hash table with disjoint path routing [15]. It provides redundant storing of data with configurable k to increase reliability and resistance to data tampering at the cost of increased storage and bandwidth requirements. The basic idea is similar to the sketched *k-set* approach to store data at k positions in the ID-space.

However, we are still in the process of designing a cheat-proof spatial index we can use to efficiently perform range queries to request the state of the game world in a certain area.

5. RELATED WORK

Proposals to realize decentralized distributed MMVEs have been around for quite some time [8]. Consequently, the problem of cheating received attention [17]. In [19] the term cheating was defined and a classification of types of cheating was developed. This classification is not specific for P2P-based MMVEs and it does not fit the five types of cheats we dealt with. They are a subset of the classified cheats but they do not show up in different categories, although they have to be treated differently.

Many approaches to deal with cheating rely on decentralized but trusted coordinators [11] or referees [16, 9] able to take final decisions on players cheating. In contrast, our approach only assumes the majority of players is honest and that we can identify and if necessary exclude players using a central certification authority. This is the weakest reasonable assumptions a system of untrusted nodes can work with.

In [12], four fundamental ways to deal with cheating are presented: distributed state dissemination, mutual checking, log auditing and trusted computing. Our approach uses mutual checking when comparing world states as well as log auditing when nodes prove their world state to be correct.

Asynchronous lock-step [1] was one of the earliest completely decentralized protocols to deal with cheating. It divided game time into rounds advancing when all nodes have committed their actions for a round until finally revealing their actions. Given game rules that allow execution of one action per round only, this approach also modifies the game state according to the game rules. It is additionally able to deal with timing cheats where nodes delay messages or modify timestamps. However, its pessimistic algorithm slows down the progression of the game time with network latency. A player suppressing actions can delay the game progression indefinitely.

The pipelining of actions makes high latency players only slow down the responsiveness of the game and not the rate at which it advances rounds [3]. NEO [7] and SEA [2] introduced additional timing bounds for players to provide their actions for a round, so players delaying or suppressing actions cannot slow down the game progression. A majority voting scheme is used to decide on the actions that have arrived in time.

None of these approaches showed how to map rules of realtime virtual worlds to a round-based system. We question the applicability of this scheme when it is not possible for update loops of computers to run synchronized. Although committing an action first and revealing it later is useful to deal with timing cheating, it introduces a two round response time to player actions. As all nodes must be able to reveal their moves within two rounds, the round length must be chosen to cover a reasonable network delay. So either responsiveness suffers or higher latency nodes are excluded from playing the game.

We believe optimistic simulation techniques like Timewarp [13] are much better suited to provide responsiveness under changing network delays and based our approach on Timewarp. An event-based realtime simulation as proposed in [6] can provide consistent world states on all nodes. Furthermore, approaches to detect timing cheats in optimistic simulations have already been proposed [4]. They might be combined with our approach to protect agains state modifications and timing cheats as well.

6. CONCLUSION

Our contribution in this paper is threefold. First, we have shown how to map typical game rules of MMOGs to an event-based realtime simulation. Using a Timewarp-based optimistic simulation approach, we can ensure responsiveness and eventual consistency. Since only player actions are exchanged, the world state evolves according to the game rules and cannot be tampered with. We have shown how four other kinds of cheats, executing forbidden actions, forging actions of honest players, sending conflicting actions, and suppressing events can be dealt with.

Second, we have shown how to obtain the correct initial world state when joining the world. A combination of vector clocks and physical clocks is used to decide whether a state is valid or not. The majority of valid world state provides the correct world state if the majority of nodes is honest.

Third, we have sketched how to make our broadcast-based approach scalable using a representative subset of nodes with size k to store events. This *k-set* receives all events and decides on the validity of an event when inconsistencies are detected. A reliable and secure storage persistently stores the world state and provides the initial state to calculate the current state. It uses redundant storing to provide the necessary resistance against state tampering by cheaters. We have shown how the same five types of cheats can be dealt with using the scalable approach.

However, the overhead to send all events to the *k-set*, to detect inconsistencies, and to retrieve missing events from the *k-set* is still considerable. Current bandwidth capacities might not be big enough to realize our scalable approach. However, the overhead is constant only depending on k. It does not depend on the number of nodes, so it is inherently scalable in contrast to the broadcast-based approach. With rising bandwidth capacities, it is only a question of time until the necessary capacity is available.

The mechanism to calculate the correct world state after a cheater has been detected also exchanges a lot of messages. However, we argue that if a mechanism is in place that successfully detects cheating and the punishment is exclusion from the game, soon there will only be a few people trying to cheat.

Some open questions remain. We have not yet analyzed the influence of churn on the reliability of the *k-set* and the reliable storage. As nodes join and leave, other nodes might take over the events stored in the *k-set*. This gives more nodes the opportunity to tamper with the stored events, to create new events or to suppress events. We also need a mathematical model to quantify the obtainable reliability under these circumstances so we can make reasonable choices for k. We have to develop a way to perform locality-based queries for retrieving the world state in certain areas from the secure storage. Finally, mechanisms are necessary to deal with timing cheating, for our approach to be applicable in practice.

7. REFERENCES

[1] Nathaniel E. Baughman, Marc Liberatore, and Brian Neil Levine. Cheat-Proof Playout for Centralized and Peer-to-Peer Gaming. *IEEE/ACM Trans. Netw.*, 15(1):1–13, 2007.

[2] Amy Beth Corman, Scott Douglas, Peter Schachte, and Vanessa Teague. A Secure Event Agreement (SEA) protocol for peer-to-peer games. In *The First International Conference on Availability, Reliability and Security (ARES)*, pages 34–41. IEEE Computer Society, 2006.

[3] E. Cronin, B. Filstrup, and S. Jamin. Cheat-Proofing Dead Reckoned Multiplayer Games. In *International Conference on Application and Development of Computer Games*, January 2003.

[4] Stefano Ferretti and Marco Roccetti. AC/DC: an Algorithm for Cheating Detection by Cheating. In *Proceedings of the 2006 international workshop on Network and operating systems support for digital audio and video*, NOSSDAV '06, pages 23:1–23:6, New York, NY, USA, 2006. ACM.

[5] Richard M. Fujimoto. *Parallel and Distributed Simulation Systems*. Wiley Interscience, New York, January 2000.

[6] Immaculada García, Ramón Mollá, and Toni Barella. GDESK: Game Discrete Event Simulation Kernel. In V. Skala, editor, *Journal of WSCG*, volume 12, Plzen, Czech Republic, February 2004. UNION Agency - Science Press.

[7] Chris GauthierDickey, Daniel Zappala, Virginia Lo, and James Marr. Low Latency and Cheatproof Event Ordering for Peer-to-Peer Games. In *NOSSDAV '04: Proceedings of the 14th international workshop on Network and operating systems support for digital audio and video*, pages 134–139, New York, NY, USA, 2004. ACM.

[8] Laurent Gautier and Christophe Diot. Design and Evaluation of MiMaze, a Multi-Player Game on the Internet. In *ICMCS*, pages 233–236, 1998.

[9] Josh Goodman and Clark Verbrugge. A Peer Auditing Scheme for Cheat Elimination in MMOGs. In *NetGames '08: Proceedings of the 7th ACM SIGCOMM Workshop on Network and System Support for Games*, pages 9–14, New York, NY, USA, 2008. ACM.

[10] Jehn-Ruey Jiang, Yu-Li Huang, and Shun-Yun Hu. Scalable AOI-cast for Peer-to-Peer Networked Virtual Environments. *Distributed Computing Systems Workshops, International Conference on Distributed Computing Systems*, 0:447–452, 2008.

[11] Patric Kabus and Alejandro P. Buchmann. Design of a Cheat-Resistant P2P Online Gaming System. In Kevin Kok Wai Wong, Lance Chun Che Fung, and Peter Cole, editors, *DIMEA*, volume 274 of *ACM International Conference Proceeding Series*, pages 113–120. ACM, 2007.

[12] Patric Kabus, Wesley W. Terpstra, Mariano Cilia, and Alejandro P. Buchmann. Addressing Cheating in Distributed MMOGs. In *NETGAMES*, pages 1–6. ACM, 2005.

[13] Martin Mauve, Jürgen Vogel, Volker Hilt, and Wolfgang Effelsberg. Local-lag and Timewarp: Providing Consistency for Replicated Continuous Applications. *IEEE Transactions on Multimedia*, 6:47–57, 2004.

[14] Gregor Schiele, Richard Sueselbeck, Arno Wacker, Joerg Haehner, Christian Becker, and Torben Weis. Requirements of Peer-to-Peer-based Massively Multiplayer Online Gaming. In *Proceedings of the Seventh International Workshop on Global and Peer-to-Peer Computing*, pages 773–782, Rio de Janeiro, Brazil, May 2007. IEEE.

[15] Sebastian Schuster, Arno Wacker, and Torben Weis. Fighting Cheating in P2P-based MMVEs with Disjoint Path Routing. *Electronic Communications of the EASST*, 17, 2009.

[16] Steven Webb, Sieteng Soh, and William Lau. RACS: A Referee Anti-Cheat Scheme for P2P Gaming. In *NOSSDAV'07: 17th International workshop on Network and Operating Systems Support for Digital Audio & Video*, pages 37–42, 2007.

[17] Steven Daniel Webb and Sieteng Soh. Cheating in networked computer games - A review. In *DIMEA '07: Proceedings of the 2nd international conference on Digital interactive media in entertainment and arts*, pages 105–112, New York, NY, USA, 2007. ACM.

[18] Torben Weis, Arno Wacker, Sebastian Schuster, and Sebastian Holzapfel. Towards Logical Clocks in P2P-based MMVEs. *Electronic Communications of the EASST*, 17, 2009.

[19] Jeff Jianxin Yan and Brian Randell. A Systematic Classification of Cheating in Online Games. In *NETGAMES*, pages 1–9. ACM, 2005.