Design of a Cheat-Resistant P2P Online Gaming System

Patric Kabus Alejandro P. Buchmann
{pkabus,buchmann}@dvs1.informatik.tu-darmstadt.de
Databases and Distributed Systems Group
Technische Universitaet Darmstadt, Germany

ABSTRACT

Since the huge success of online games and especially that of so called *Massively Multiplayer Online Games (MMOGs)*, the prevention of cheating has been a primary concern of game providers. This is also the main reason why Peer-to-Peer (P2P) architectures are not widely used in this area, although they could offer significant cost savings. Letting player nodes handle game-relevant logic or data means giving away control and opening up additional possibilities for cheaters. In this paper we present a P2P architecture which provides countermeasures against cheats that are inherent to these systems.

1. INTRODUCTION

Despite the tremendous growth of subscriber numbers—currently several millions for the market leaders [38]—the underlying network architecture for MMOGs has remained the same since their beginnings. All of today's successful MMOGs still follow the Client/Server (C/S) paradigm, demanding amply equipped back-ends to handle the load of many users. Fulfilling the processing power, storage capacity and network bandwidth requirements plus employing the necessary maintenance personnel is cost-intensive. An example for the benefits of using P2P mechanisms can be observed by looking at the market leader, *World of Warcraft* [39]. While the game itself is a C/S system, updates of the client software are distributed through a P2P network. This way a fraction of the bandwidth requirements at the server-side are imposed on the clients, reducing costs and maximizing revenues of the game provider. Generally, P2P systems could reduce costs by shifting processing, storage or bandwidth demand from the server to the client.

However, offloading the burden doesn't come without a price. Letting a client handle game-relevant tasks also takes away control from the game provider. Far worse than faulty clients, which could accidentally loose data or perform incorrect computations, are clients that actively try to exploit their new responsibilities. With the ever-growing success of network multiplayer games, the cheating problem has moved into the focus of the gaming industry [19, 34]. As long as games didn't have any network functionality, cheating was hardly relevant. But as soon as a part of the game runs on a remote machine, trustworthiness becomes an issue. While playing local area network games or games with only a small number of players, one could easily restrict the participants to personally known and trusted people. However, this is not feasible for large-scale Internet multiplayer games; especially subscription-based MMOGs with persistent game worlds are at risk. Successful cheating attempts have lasting effects on a persistent world, posing a long term threat to its balance and thus to the fairness of the game. Fairness is critical for keeping players motivated to maintain their subscriptions.

In this work we present a P2P gaming system design that addresses cheating attacks which are inherent to these systems. We

- identify the relevant attacks and their possible impact
- present a system design based on selective replication that addresses these attacks
- analyze concrete attack scenarios and countermeasures
- evaluate a prototype implementation

The following section identifies fundamental building blocks and assembles them into a workable system. To enable a more detailed understanding, section 3 categorizes attacks and explains how the system counteracts them. The proposed system design has been implemented as a prototype which is evaluated in section 4. Related work is presented in section 5, while section 6 concludes the paper and points out areas of future research.

2. SYSTEM DESIGN

In this section we discuss fundamental decisions that have to be made when designing a cheat-proof distributed gaming system. These choices make up the cornerstones of the proposed gaming system.

2.1 Failure Model

Some people try to exploit any kind of security flaw for their own profit while others amuse themselves by harassing other players and causing arbitrary havoc. Unfortunately, these attacks introduce a byzantine failure model since a malicious node in a P2P system may exhibit any kind of unwanted behavior. Handling byzantine failures introduces an inherently high complexity. Thus, we believe that cheating countermeasures have to be taken into account in the early design stages of a system and cannot simply be added as a module later.

DIMEA '07 Perth, Western Australia

2.2 Cheating Attacks

In the following we will identify the cheating attacks that our system addresses. We will discuss the impact of these attacks and present appropriate countermeasures.

2.2.1 What to address

Yan and Randell [40] give a rather comprehensive overview of cheating in online games and define a cheating taxonomy. Many of the cheating attacks listed in that paper also apply to traditional Client/Server architectures and countermeasures have already been developed in this context. In contrast, we will focus on attacks that are inherent (but not exclusive) to a P2P-based gaming system. These are

Exploiting Misplaced Trust The software as well as game state data are stored locally on players' machines. This makes them susceptible to any kind of malicious manipulation.

Exploiting Lack of Secrecy Without further protection a peer is not only able to access all data that is stored locally but it may also disclose it to other clients.

Collusion Malicious peers within the system may collude in performing cheats.

In contrast to Yan and Randell, we consider *Collusion* to be orthogonal to the other attacks. Thus, both kinds of cheats, *Exploiting Misplaced Trust* and *Exploiting Lack of Secrecy*, can be performed either by a single player or by multiple colluding players.

2.2.2 Impact on Gameplay

Exploiting Misplaced Trust by manipulating game logic and data is highly relevant to any kind of game. In a Role-Playing Game (RPG) cheaters may make their virtual character invincible by altering its attributes. In a racing game they can raise the speed of their vehicle, while in a strategy game they can provide themselves with unlimited resources. It is obvious that for any kind of game a cheater greatly benefits from altering certain game state data.

The relevance of *Exploiting Lack of Secrecy* is heavily dependent on the kind of game. In a strategy game for example, knowing the position of enemy troops can give a crucial advantage over other players. But in a racing game where driving skills are the key to success, information about competing drivers may be rather uninteresting.

2.2.3 Countermeasures

The basic principle to address arbitrary manipulations of game data is to replicate the game state among multiple nodes instead of having to trust a single one. A computer game is basically a finite state machine which transforms its internal game according to user input. Thus, a distributed game that replicates its state can be classified as a *Distributed State Machine* [35]. If each replica starts in the same initial state and processes events in the same order, all of the correct working replicas will eventually reach the same state. The correct result is determined by a majority voting and thus we can cope with up to b replicas exposing byzantine behavior if there are at least $2b+1$ replicas in total. Since both malicious and faulty nodes can be subsumed as byzantine nodes, b equals the number of faulty nodes f plus the number of malicious nodes m. In [26] we sketched out different ideas for countermeasures against cheating in distributed games. In this paper we flesh out two of the presented concepts (namely *Mutual Checking* and *Log Auditing*) and integrate them into our system.

Our distributed state machine approach partitions the game world into smaller regions to distribute the computational load (see section 2.4 on how this can be done). Each region is assigned a group of nodes which maintains the state of this region. Nodes responsible for managing the state of a region are called *Region Controllers (RCs)*. Every RC maintains the whole state of a region, i.e. the state is replicated on all RCs of the group. A player's node connects to all members of the group which is responsible for the region in which the player's avatar is located. The node runs a client software which takes input from the player and sends it as a request to the whole group. After processing the request each RC of the group sends its state update back to the player's node. The node takes as the correct answer the one which has been sent by the majority of the RCs. The answer contains information about changes in the game region and the client displays it to the player. As long as faulty or malicious RCs are outnumbered by correct ones, the correct game state will be displayed. Now the player can react to changes in the environment and start over the process by generating new input. Every node in the P2P system can act both as a client and as an RC. Note that we do not require a group of RCs to exchange messages in order to agree on the state of a region. Instead, we completely rely on the voting mechanism on the client side and thus avoid the message overhead.

Before we address the *Lack of Secrecy* problem, we must make a distinction between two different cases. Either an unauthorized node tries to access data that is stored locally or an authorized but malicious node discloses data to an unauthorized one. The former case can be addressed, but we will show that the latter cannot. In order to prevent unauthorized local access, confidential data may only be stored locally if it is encrypted or not stored locally at all. If the data is encrypted, the local node can only store it but cannot perform computations on it. The node may serve as a data cache for authorized nodes or as a simple relay. The system we propose can prevent storing data on unauthorized clients if necessary (again, see section 2.4). The case of an authorized node disclosing confidential data to unauthorized ones cannot be prevented by technical means. Think of a digital version of a card game like *Bridge* [1], where two players form a partnership but they do not know each other's cards. Players are authorized to know the cards in their own hand but they may not disclose them to their partner. Even if the system was able to prevent the exchange of data between the nodes of a team, it couldn't stop players from talking directly to each other or from using a phone. Fortunately, only a very limited range of game designs (namely team-oriented strategic games, where team members may not share information) is affected by this kind of attack.

In addition to the countermeasures above, the system re-

quires that every message is signed by the sender. Every recipient keeps a log of all received messages for a certain period of time and signs it. This way every node can prove which message it received within that period. By sending the log to a trusted service run by the game provider, a node can prove that it received a falsified message.

2.3 Replication Issues

Our replication approach raises three main issues: replica placement, consistency and update propagation.

2.3.1 Replica placement

Since the motivation for distribution is reducing costs, replicas will be hosted on the players' machines. To reduce complexity and to avoid potential security breaches, the placement of replicas will be managed by super-peers run by the game provider. The effort of managing replicas is considerably lower than managing the whole state of possibly huge game worlds. The hybrid approach—distributing game logic among players' machines and providing trusted super-peers only for specific tasks—will reduce the complexity significantly while largely retaining the cost savings.

2.3.2 Consistency

In section 2.2.3 we state that the correct game state will be determined by a majority voting. Consequently, the consistency protocol applied in the system may not leave replicas in different states. Many different consistency models have been presented in the literature, [36] contains a rather comprehensive overview. We chose *Sequential Consistency* [28], since it provides us with the necessary common global state. *Linearizability* [25] is slightly stronger, but also more expensive in terms of worst-case response time [6]. *Causal Consistency* [4] is weaker than Sequential Consistency and guarantees a common global state only in special cases. Even weaker consistency models are not applicable since they do not maintain equal states on replicas. There also exist variations of common consistency models and implementations more specific for distributed games. One example is *Rendezvous* [16], a decentralized consistency management mechanism that is targeted at multiplayer games in high latency environments. A key feature is that it always maintains a certain degree of inconsistency in order to improve response time. Unfortunately, the permanent inconsistency would break our voting scheme.

On the client-side we have to take care of the delay caused by the request/reply round-trip. Since the quality of the players' game experience is affected by this delay [33], it should be hidden from the player using techniques like *Dead Reckoning* [32] or *Pre-Reckoning* [20].

2.3.3 Update propagation

The approaches to update propagation can be differentiated between *Active Replication* and the *Primary-Backup Approach* [11].

With the Primary-Backup approach, sequential consistency can be achieved easily. For every data item exists a primary replica (PR) which serializes all operations on that item. Unfortunately, a node may act maliciously or faulty only after it has become a PR. Neither can this be predicted nor is it trivial to determine that the PR is actually misbehaving. But even if it could be detected, the election of a new PR would face again the problem of determining a well-behaved node.

Active replication avoids these problems, since all replicas are equal. The challenge is that due to network conditions, client requests sent directly to the replicas may reach them in different orders. But sequential consistency requires that all requests sent by a single client are processed in the order they were issued by that client and that there is some global sequence of all requests. To achieve this, we can exploit the fact that all games break down simulation time into discrete slices (we will call them *frames*). Every client may issue a single request per frame and transmit the frame number within each request. This way the sequence of requests sent by a single client can easily be maintained. To achieve a global ordering of client requests within a frame they can be processed according to a fixed and unique client id. Note that we need only some global order that retains the local order of each client. A perfect global order (called *Strict Consistency*) is neither necessary nor feasible in a distributed system [37].

The order of request processing can be changed every frame (e.g. rotating the order). Otherwise the player with the lowest id is always the first to act, which would probably affect fairness. The size of a frame determines how often the game world gets updated and thus the pace of action. Fast-paced games need smaller frame sizes, while slower games allow for larger frames.

2.4 Partitioning of the Game World

Dividing the game world into different regions and distributing them among a set of nodes is very easy, as long as there is no interaction between regions. Every RC group that hosts a region manages all the game objects within it and performs all the logic necessary for the region. Not allowing interactions between regions makes synchronization between RC groups unnecessary. But non-interacting regions are effectively separate game worlds, so some sort of interaction must be possible. A simple but effective kind of interaction is the transfer of objects between regions. Imagine a player that leaves a region (e.g. through a door) and reappears in an adjacent region. Just transferring object data between regions doesn't imply the need of synchronization between them.

The notion of having separated regions which restrict interactions between them to object transfers fits closely the concept of *instances* in todays online games. An instance is a separate region (e.g. a dungeon) of the game world that is only shared by a small group of players (usually between five and forty). Different groups of players can occupy the same region, but each group gets their own copy of the region from scratch. This way a group can explore a region, kill monsters there and loot the treasures without being disturbed by other groups. For the game provider, an important advantage of this concept is that an instance can be spawned on whichever server currently has enough free resources. The limited amount of players within an instance makes its resource consumption predictable and thus supports proper load balancing. Nearly every success-

ful MMOG today makes use of this concept. Some, like *World of Warcraft* [39], use instances for special parts of the world, while others, like *Guild Wars* [2], use instances for nearly all areas.

For the rest of this paper, we focus the view on a single instance and region respectively and how it is managed. Since object transfer is the only interaction between regions, we do not need synchronization between them. If the game design requires that players may not have read access to all data of the region in which their avatar is currently located, a region may not be spawned on that player's node (i.e. the node may not become an RC for that region). If a player's node is already an RC for a certain region, it is not allowed that this player joins a group which is currently located in that region. To guarantee adherence to these rules, the region allocation is managed by a trusted peer (see next section).

2.5 Additional Services

We have outlined the basic principles of our proposed system. In order to create a fully functional system, we need two additional services.

The *Management Service* is the first contact point for players joining the game, since it maintains a list of all RCs which are currently in the game. It informs a newly joined client about the RCs that are currently responsible for the region of the player's avatar. If the avatar enters a new region, the Management Service will provide information about the new RC group. Since the player's node may also act as an RC, the Management Service will put a newly joined node into a pool of currently unused RCs. Whenever there is the need for an additional RC (e.g. because one of the already deployed RCs left the game or failed), an unused RC will be chosen randomly from the pool for this task. Random selection of RCs makes it very unlikely that a group of colluding malicious nodes acquire the majority within an RC group. Obviously, this is dependant on the number of RCs within a group, i.e. the degree of replication. An important fact is that nobody can join the game without permission of the Management Service. This is necessary for enforcing any kind of payment model. Since personal data may be stored on the Management Service and because of the need for permanent availability, the Management Service should be run by the game provider.

If a game world is persistent, a *Persistence Service* is necessary to create a periodical backup of its state. This is necessary in case the system crashes or needs to be shut down for maintenance purposes. Moreover, if a region of the game world is currently empty, i.e. there are no avatars in it, the region can be shut down and the corresponding RCs can go back to the pool of free RCs. In order to create a snapshot of the current region state, all RCs of that region send the changes since the last backup to the Persistence Service. It determines the correct state by choosing the one which holds the majority. Every time players leave, their avatar's data can be sent to the Persistence Service. Whenever they join again, their data can directly be sent to the responsible RCs. This makes it unnecessary for RCs to store data of avatars which are currently not in the game. The Persistence Service may be provided by the game hoster or could be itself a P2P-based system running on players' nodes.

3. SCENARIOS

In order to fully understand how the proposed system detects and prevents cheating attacks, we will now have a look at the different scenarios that may occur during operation. We start with the ideal case where only benign nodes are present. We then examine the possible attacks malicious nodes can perform depending on which role they play, either an interface client or a Region Controller.

3.1 Normal Operation

We first describe the case where no cheaters are present to show how the system works during normal operation. We assume that the system is up and running and start at the point where a player joins the game.

3.1.1 Logging into the system

Whenever a player wants to join a game session, the client software connects to the Management Service which is responsible for authenticating the player. On successful authentication, the state of the player's avatar is retrieved from the Persistence Service.[1] This state contains information about the avatar's current location within the game world, e.g. the place where the avatar was located when the player quit playing the last time. The avatar's state is transmitted to the responsible RC group. Additionally, the network addresses of the responsible RCs are sent to the player's client. Finally, all RCs transmit the state of the avatar together with the necessary environmental information (e.g. the avatar's immediate surroundings) to the player's client. Since all RCs are honest the player should receive the same data from all RCs. Now both sides, the RCs and the player's client, possess all the necessary information to enable the player to start playing the game.

3.1.2 Playing the game

As soon as the client has displayed the received information, the player will want to start interacting with the game world. In order to do so, the client software translates the player's input into action requests (e.g. commands like *move*, *attack*, etc.) and sends them to each RC on the list which was received during log in. All RCs receive the same action request, process it and update their local game state accordingly. Since game state transitions are deterministic, all RCs should arrive at the same state as long as they process the requests in the same order. How this can be achieved has been discussed in section 2.3. After the game state transition has been performed, the new state has to be transmitted to all interested clients.All RCs send an update packet to each client that contains only the changes that are relevant for this client. Since the state on all RCs is the same, all updates received by a single client are equal. The client incorporates the update into its local game state and after that it is ready to issue the next action request.

[1]In a commercial game with subscription fees, the authentication service may check at this point whether the player has paid the fees. If not, it may reject the login.

3.1.3 Logging out of the system

When the player leaves the game, the state of the avatar has to be written to persistent storage so it can be restored whenever the player decides to join again. The client issues a log-out command to all RCs. Upon receiving this command, the RCs send the current state to the Persistence Service. Since the state on all RCs is equal, the service receives the same avatar state from all RCs and writes it to persistent storage. Usually, it is desirable to perform a backup not only at the time the player leaves the game but also on a periodical base. Thus, the RCs transmit game state changes at certain intervals to the Persistence Service.

3.2 Cheating Scenarios

This section gives an overview of the different cheating scenarios that may be encountered and discusses how the system deals with them. From now on, we assume that all messages exchanged within the system are signed by the sender. Messages which are not properly signed won't be accepted. In many cases, a signature enables honest participants of the system to prove the origin of cheating attempts. The possibility to track down a cheater will make such attempts significantly more risky. Since any communication is done via network messages, cheating is done either by sending forged messages, sending aberrant messages to different receivers or omiting messages. All the countermeasures assume that there is a majority of well-behaving RCs. The higher the degree of replication, i.e. the more RCs are responsible for a region, the lower the chance that the misbehaving ones will attain the majority. We believe that having less than half of the nodes being malicious or faulty is a realistic assumption.

3.2.1 Forged messages

We consider a message to be forged whenever it deviates from a message that would have been sent by an honest sender. Either the message was forged directly or it was created as a result of a state manipulation. For example, if the local state of an RC has been manipulated it may send an update to its clients that differs from an update sent by an honest RC. Note that the incentive to send forged messages is rather low since the issuer can always be identified through the signature.

Clients send forged action requests. A manipulation of a client's local game state is futile because it never transmits its state to anyone else. Since it only sends requests to its RCs, its possibilities to cheat are rather limited. An action request may contain raw player input (e.g. mouse clicks, button presses) or more abstract commands (e.g. "move to (x,y)", "attack object z"). In any case the RCs will perform a sanity check on the request: the state transition caused by the request must be legal according to the rules of the game. If a request would cause an illegal state transition it is not processed by honest RCs. Since a client sends only action requests for itself and may not directly issue state changes, it is hardly possible for clients to collude in a cheating attempt. However, it is possible for a client to collude with an RC. A colluding RC would simply accept the illegal action request and consequently perform an illegal state transition. But from an external point of view it is irrelevant whether the RC accepted an illegal action request or simply manipulated the state itself. All that can be seen from the outside is an RC issuing illegal state updates since its internal state is corrupt.

RCs send forged update and backup messages. In contrast to clients, RCs transmit changes of the game state periodically to others, namely as updates to clients and backups to the Persistence Service. Since all honest RCs are at the same state, the messages sent by them are equal. A message sent by a malicious RC can easily be detected by comparing it to those sent by the other honest RCs. Even multiple RCs (colluding or not) will not be able to falsify the game state as long as our basic assumption holds true. Messages that deviate from the majority can simply be dropped.

3.2.2 Aberrant messages

In this section we will see that attacks may be performed even by sending legal but deviating messages to RCs. Aberrant messages can only be used to disturb the game but not to gain any advantage over other players. Additionally, senders of aberrant messages can be identified through the signature and punished.

Clients sending aberrant action requests. A client may send different legal action request to different RCs. Since each of them will accept the request, they will eventually reach different states and thus send different updates and backup messages. Upon receiving the deviating replies the attack can be detected and the malicious client can be identified and banned.

A special case where a malicious client colludes with multiple RCs shows why only absolute majorities should be accepted, even if in other cases a relative majority would be sufficient. A client may send deviating requests to honest RCs, so that each of them reaches a different state. The colluding RCs can now acquire a relative majority by agreeing on a certain falsified state. But as long as they cannot acquire the absolute majority, their update and backup messages will not be accepted. Instead, the deviating replies from the honest and the malicious RCs will reveal the attack.

RCs sending aberrant update and backup messages. Malicious RCs cannot launch a successful attack until they hold the absolute majority. Since both, the client and the storage system, will only then accept their messages.

3.2.3 Omitted messages

By not sending a message, it is not possible to alter the game state and thus a player cannot gain any advantage. Thus the incentive for this attack may be low. But since a single client can disturb the game this way, we will have a short look at this attack. If a client sends a legal request to one part of the RCs and omits the message for the rest, whichever group was smaller will go out of sync (since the larger group holds the majority). The affected RCs need to recover the state from the Persistence Service or will be replaced by new ones. Depending on how much time is needed for this action, a delay could be visible within the

game. RCs that do not send update or backup messages do not have any effect. Since the majority sends correct messages, the voting procedure yields the correct result.

3.3 Correcting/Replacing Misbehaving RCs

RCs that send updates deviating from the majority need to be corrected or replaced. If an RC is out-of-sync simply because of a delayed message, it can recover by itself after it received that message. But RCs that continue to send deviating updates are probably malicious and need to be replaced. The process of replacing an RC may be initiated by the clients which can easily detect differences in the received updates. If a majority of clients claims that an RC is out of sync, the Management Service can replace it with a fresh one from the pool. Alternatively, the Persistence Service may detect deviating RCs and initiate the replacement. In order to serve as an RC, a node needs to learn the current state of the region. Again, either the remaining RCs can provide the current state (as usual, the new RC uses the state the majority votes for) or the Persistence Service after it received its periodical update.

4. EVALUATION

In order to evaluate the feasibility and performance of our approach, a prototype has been implemented. The implementation includes clients, region controllers, the Management Service and the Persistence Service. The game world consists of fixed size regions and mobile game objects which are able to move within regions and change regions. The focus of our evaluation lies on the consistency among the replicas. An important factor here is the size of a frame (see section 2.3.3). The smaller a frame is, the more responsive a game will be since more request/update cycles are performed per time. However, smaller frame sizes bear a danger for consistency. As stated in section 2.2.3, RCs do not compare their current state to avoid a considerable message overhead. This also means that RCs do not exchange messages to synchronize the frame advancement. In addition to the overhead, this synchronization would allow single misbehaving nodes to delay or even block the advancement. Unfortunately, without frame synchronization single RCs may lag behind or advance too fast. Alternatively, the advancement of frames on the RCs can be coordinated by synchronizing their local clocks. In order to do so, a standard NTP client was deployed on each node. The deviation error introduced by using NTP over the Internet is usually about 20ms or less according to [30]. Minor frame inconsistencies may still occur because of this deviation. However, in the scenarios presented in this paper these inconsistencies can be compensated for quickly without visibly affecting gameplay (see section 3.3).

The variables in our evaluation were the network delay, the number of clients per region, the number of RCs per region and the frame size. For the network delay we considered two scenarios, Local Area Network and Internet. In order to emulate realistic Internet delays we used a traffic control mechanism built into the kernel of our nodes' operating system. It offers the possibility of delaying network packets according to a predefined distribution. We chose a distribution that resembles the average Internet delay that can be observed within North America according to the *Internet Traffic Report* [3]. We considered the cases of 5 and 26 clients for a single region (typical size of small and medium sized groups within an instance dungeon of current MMOGs) which were controlled by 5 or 11 region controllers. The number of nodes was also limited by our testing environment.[2] We evaluated different frame sizes, starting at 100 milliseconds. A scenario with a certain combination of values for the variables named above is considered feasible if only a minority of RCs suffers from recoverable short term inconsistencies or no inconsistencies occur at all. Since the majority of RCs always maintains the correct state, the inconsistencies are not visible to the players.

Due to space limitations, we present only the most relevant test results. These come from our reference scenario and the scenarios with the smallest feasible frame lengths we could achieve. For the complete evaluation including 17 test scenarios see [18].As the reference scenario, we used 5 clients and 5 RCs. This means that in our testing environment each node runs only a single client/RC combination (as it will be in a real world deployment). The additional services run on separate nodes. In this scenario, we avoid measurement errors due to overloading the rather slow testing machines. Since nowadays even low-cost PCs have dual-core CPUs significantly faster than our testing machines and RAM in the size of gigabytes, we expect the average gamer's PC to greatly outperform our testing machines. The evaluation shows that in the reference scenario under LAN conditions, a frame size of 200ms can be achieved. In the reference Internet scenario, we could still achieve frame sizes as small as 300ms. In both scenarios with 26 clients and 11 RCs the feasible frame size extended to 500 ms. Running multiple clients and RCs on a single node generates load peaks which may have a negative impact on the error rate. Especially in a LAN environment, smaller frame sizes should be feasible. Figure 4 shows the four feasible scenarios described above. The last row shows the probability of a temporal inconsistency per frame and RC which could be resolved by correcting or renewing the affected RC. Note that during the scenarios including eleven RCs, the number of inconsistent RCs never exceeded two (one in the five RC scenario). This means that even if cheaters could guess the exact point in time where these inconsistencies occur, they would need to control four out of eleven RCs to interfere with the voting procedure.

The evaluation shows that games which do not rely on high responsiveness can be realized with this architecture. Even slower paced action games, where the delay can be hidden from the player by client-side prediction are possible. Not surprisingly the architecture is not suited for fast-paced action games.

In addition to the frame size evaluation, we performed calculations on the maximum number of clients that can be supported within a single region. Under the assumption that player nodes are connected through common ADSL lines, the possible number of clients is about 25. This is still sufficient for player groups commonly seen in the instances described in section 2.4. However, if we consider more recent but already available standards like VDSL, which supports synchronous connections with up to 26 Mbit/s, several hun-

[2]Seven PCs equipped with an Athlon XP 2000+ CPU and one GB of RAM.

Scenario	2	7	11	16
Delay	LAN	LAN	Inet	Inet
# Clients	5	26	5	26
# RCs	5	11	5	11
Frame length	200ms	500ms	300ms	500ms
% Incons.	0.009	0.004	0.006	0.063

Figure 1: Selection of test scenarios

dreds of clients are possible within a single region. Assuming that the game world might consist of dozens or hundreds of these regions, we arrive at total player numbers that exceed those of today's MMOGs. The number of simultaneous regions is mainly limited by the capacity of the super-peers that provide the management and persistence services. For a detailed discussion of the underlying figures and equations we, again, refer to [18].

5. RELATED WORK

FreeMMG [14, 13] is a hybrid between Peer-to-Peer and Client/Server architecture and similar to our approach. While a server part is responsible for managing subscriptions, authentication and storing backups of the virtual world, the game itself is running in a distributed fashion on the clients. The game world is split into segments and segments are replicated on the nodes of the players. Unlike the system presented in this paper, FreeMMG stores a segment's state on the node of the players within that segment. This opens up the possibility of disclosing secret information directly to the players. The replicas use a lockstepping mechanism to stay synchronized which allows single malicious nodes to break the whole synchronization. Moreover, many aspects of the system remain unclear. First, there is no systematic classification of attacks with an explanation of how the system counteracts them. Only very few cheating scenarios are considered briefly. It is also not clear how the correct game state is determined in the presence of cheaters. Finally, the authors haven't found an appropriate consistency protocol yet. Although central parts are missing, a prototype of the system has been implemented. How this implementation is able to function in the presence of these gaps is not explained.

Another hybrid system that claims to provide cheat resistance is published in [17]. Again, the game world is split up into smaller regions which are manageable by player nodes. This time, the primary backup approach discussed in section 2.3.3 is used. Unfortunately, the paper stays on a very abstract level without providing any details about state synchronization, attack scenarios or correct state determination.

An interesting approach is presented in [31] which breaks up with the assumption that a client is inherently not trustworthy. To ensure the integrity of a client, a protection mechanism is embedded into the software. In order to prevent an attacker from bypassing the protection, the protection code will be constantly changed within short intervals. The client has to download always the latest version of the code in order to be allowed to play. The authors claim that breaking the protection within the small period when it is active is not feasible. Since this approach is orthogonal to the system presented in this paper, they could be combined to provide a higher level of protection.

There are also many publications on other kind of attacks less relevant to MMOGs than those addressed by our system. Baughman et. al. [7, 8] propose a scheme that uses a lockstepped commitment protocol to prevent cheats on the protocol level. The NEO protocol [22] was developed as an improvement to the one presented above. It addresses a broader range of cheats while at the same time reduces latency but still addresses only cheats on the protocol level. Another approach on a similar level is AC/DC [21], which addresses cheats based on game event timing. Buro [12] presents a server-based architecture which addresses the *maphack* cheat popular in Real-Time Strategy Games (RTS). Chambers et al. [15] show that this kind of attack can also be addressed in a Peer-to-Peer architecture. There exist more proposals for distributed gaming architectures like Mercury [10], SimMud[27], MiMaze [29, 24, 23], Colyseus [9] and the approach implemented in Kosmos [5]. However, they do not explicitly address the problems originating from cheating.

6. CONCLUSION

In this paper we have presented the design of a Peer-to-Peer gaming system that provides appropriate countermeasures against certain cheating attacks. It prevents malicious manipulations of the game state although this state is stored on the players' nodes. Moreover, the system distributes the game state in a way that prevents players from directly accessing data that is of direct relevance to them (e.g. information about their avatar's immediate surroundings). The approach relies on active replication of the game state on players' nodes as well as on services provided by super-peers. These super-peers remain under the control of the game provider, thus providing means to enforce a subscription-based payment model. We have analyzed possible attacks and how our system can handle them, either by preventing them completely or by detecting them and taking appropriate measures. Finally, we have shown by evaluating a prototype implementation, that realizing enjoyable games with this approach is feasible. The next step will be an evaluation on a larger scale, including more nodes within a single region or instance. Since our current hardware resources are limited, we consider using a simulation-based approach.

7. REFERENCES

[1] Bridge. `en.wikipedia.org/wiki/Contract_bridge`, 2007.

[2] Guild Wars. `www.guildwars.com`, 2007.

[3] Internet traffic report. `www.internettrafficreport.com`, 2007.

[4] M. Ahamad, P. W. Hutto, G. Neiger, J. E. Burns, and P. Kohli. Causal memory: Definitions, implementation and programming. Technical Report GIT-CC-93/55, Georgia Institute of Technology, 1994.

[5] M. Assiotis and V. Tzanov. A distributed architecture for MMORPG. In *Proceedings ACM NetGames '06*, 2006.

[6] H. Attiya and J. L. Welch. Sequential consistency versus linearizability. *ACM Trans. Comput. Syst.*, 12(2):91–122, 1994.

[7] N. E. Baughman and B. N. Levine. Cheat-proof playout for centralized and distributed online games. In *Proceedings IEEE INFOCOM*, volume 2, pages

104–113, April 2001.

[8] N. E. Baughman, M. Liberatore, and B. N. Levine. Cheat-proof playout for centralized and serverless online games. Technical report, University of Massachusetts Amherst, 2004.

[9] A. Bharambe, J. Pang, and S. Seshan. Colyseus: A distributed architecture for online multiplayer games. In *Proceedings ACM NSDI '06*, 2006.

[10] Ashwin R. Bharambe, Sanjay Rao, and Srinivasan Seshan. Mercury: a scalable publish-subscribe system for internet games. In *Proceedings ACM NetGames '02*, pages 3–9. ACM Press, 2002.

[11] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg. The primary-backup approach. pages 199–216, 1993.

[12] M. Buro. ORTS: A hack-free RTS game environment. In *Proceedings International Joint Conference on AI '03*, 2003.

[13] F. Reis Cecin, R. de Oliveira Jannone, C. F. Resin Geyer, M. Garcia Martins, and J. L. Victoria Barbosa. FreeMMG: a hybrid peer-to-peer and client-server model for massively multiplayer games. In *Proceedings ACM NetGames '04*, pages 172–172. ACM Press, 2004.

[14] F. Reis Cecin, R. Real, M. Garcia Martins, R. de Oliveira Jannone, J. L. Victoria Barbosa, and C. F. Resin Geyer. FreeMMG: A scalable and cheat-resistant distribution model for internet games. In *8th IEEE International Symposium on Distributed Simulation and Real Time Applications*, 2004.

[15] C. Chambers, W. Feng, W. Feng, and D. Saha. Mitigating information expose to cheaters in real-time strategy games. In *Proceedings ACM NOSSDAV '05*, 2005.

[16] A. Chandler and J. Finney. On the effects of loose causal consistency in mobile multiplayer games. In *Proceedings ACM NetGames '05*, pages 1–11, New York, NY, USA, 2005. ACM Press.

[17] A. Chen and R. R. Muntz. Peer clustering: A hybrid approach to distributed virtual environments. In *Proceedings ACM NetGames '06*, 2006.

[18] F. Dautermann. Implementierung und analyse eines manipulationssicheren multiplayer online game systems. Master's thesis, TU Darmstadt, 2007.

[19] S. B. Davis. Why cheating matters - cheating, game security, and the future of global on-line gaming business. In *Proceedings of the 2003 Game Developers Conference*, March 2003.

[20] T. P. Duncan and D. Gračanin. Algorithms and analyses: Pre-reckoning algorithm for distributed virtual environments. In *WSC '03: Proceedings of the 35th conference on Winter simulation*, pages 1086–1093. Winter Simulation Conference, 2003.

[21] S. Ferretti and M. Roccetti. AC/DC: an algorithm for cheating detection by cheating. In *Proceedings ACM NOSSDAV '06*, 2006.

[22] C. GauthierDickey, D. Zappala, V. Lo, and J. Marr. Low latency and cheat-proof event ordering for peer-to-peer games. In *Proceedings ACM NOSSDAV '04*, pages 134–139. ACM Press, 2004.

[23] L. Gautier and C. Diot. Design and evaluation of MiMaze, a multi-player game on the internet. In *Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, page 233. IEEE Computer Society, June 1998.

[24] L. Gautier and C. Diot. Distributed synchronization for multiplayer interactive applications on the internet. Unpublished, 1998.

[25] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[26] P. Kabus, W. W. Terpstra, M. Cilia, and A. P. Buchmann. Addressing cheating in distributed MMOGs. In *Proceedings ACM NetGames '05*, pages 1–6, New York, NY, USA, 2005. ACM Press.

[27] B. Knutsson, H. Lu, W. Xu, and B. Hopkins. Peer-to-peer support for massively multiplayer games. In *INFOCOM '04*, March 2004.

[28] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28, Issue: 9:690– 691, 1979.

[29] E. Lety, L. Gautier, and C. Diot. MiMaze, a 3D multi-player game on the internet. In *Proceedings of the 4th International Conference on Virtual System and MultiMedia*, November 1998.

[30] N. Minar. A survey of the NTP network, December 1999.

[31] C. Mönch, G. Grimen, and R. Midstraum. Protecting online games against cheating. In *Proceedings ACM NetGames '06*, 2006.

[32] W. Palant, C. Griwodz, and P. Halvorsen. Evaluating dead reckoning variations with a multi-player game simulator. In *Proceedings ACM NOSSDAV '06*, pages 20–25, May 2006.

[33] L. Pantel and L. C. Wolf. On the impact of delay on real-time multiplayer games. In *Proceedings ACM NOSSDAV '02*, pages 23–29, New York, NY, USA, 2002. ACM Press.

[34] M. Pritchard. How to hurt the hackers: The scoop on internet cheating and how you can combat it. *Gamasutra*, 2000.

[35] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Comput. Surv.*, 22(4):299–319, 1990.

[36] R. C. Steinke and G. J. Nutt. A unified theory of shared memory consistency. *J. ACM*, 51(5):800–849, 2004.

[37] A. S. Tanenbaum and M. van Steen. *Distributed Systems - Principles and Paradigms*. Prentice-Hall, 2002.

[38] B. S. Woodcock. An Analysis of MMOG Subscription Growth. `www.mmogchart.com`, 2006.

[39] WoW. World of Warcraft. `www.worldofwarcraft.com`, 2007.

[40] J. Yan and B. Randell. A systematic classification of cheating in online games. In *Proceedings ACM NetGames '05*, pages 1–9, New York, NY, USA, 2005. ACM Press.