

# Analysis of State Exposure Control to Prevent Cheating in Online Games

Kang Li Shanshan Ding Doug McCreary  
Department of Computer Science  
University of Georgia  
Athens, GA 30602, USA  
{kangli, ding, mcreary}@cs.uga.edu

Steve Webb  
College of Computing  
Georgia Institute of Technology  
Atlanta, GA 30332, USA  
webb@cc.gatech.edu

## ABSTRACT

Cheating has become a serious threat to the online game industry. One common type of cheating is accessing game states that are not supposed to be exposed to the players. In this paper, we evaluate a few information dissemination strategies that limit the state exposure to the client, measuring the delay introduced to the players and the system resource requirements at the game server. Our measurement is based on OpenGladiator, a multi-user, real-time strategy game that is similar to Warcraft but in open-source. We found that by performing careful on-demand preloading, we can significantly reduce the unnecessary states exposed to a client without introducing any additional delay to the players. Our measurement results show that the on-demand strategy comes with a increment to the server's CPU load. However, it also significantly reduces the server's network bandwidth consumption, which is a major cost of running a game server in the current Internet.

**Categories and Subject Descriptors:** D.4.6 [Security and Protection]: Information Flow Control

**General Terms:** Measurement, Performance

**Keywords:** Online Games, Anti-Cheating

## 1. INTRODUCTION

With the increasing number of players, multi-player online gaming has become a very profitable industry[1, 2]. A particular feature that attracts many users to multiplayer online games is the involvement of and interactions with human players. However, as games now more often include human players competing with each other, *game cheating* has become a serious problem. Cheating was not a big threat to the game industry in the past when it was used against machines, but it can ruin the fun when used against human players. For game developers, cheating is a large threat to their potential income because cheaters will make games less attractive to non-cheaters. Cheating in games can be in many forms [3, 4, 5], and preventing cheating has become a challenging problem.

One common type of cheating is accessing game states that are not supposed to be exposed to the players. This includes accessing static information (e.g. a map) of the simulated environment or dynamic states (e.g. positions of the enemy's units). Knowing any of this state information is a superior advantage to a player. By exploring the static state information, a map hack allows online players to disable "fog of war" (the line of sight for each unit) so that the entire map is revealed. This allows the cheater to know areas of the map not previously explored. When dynamic state information is exposed by the cheaters, they can even monitor the enemy and strategically position their units.

One solution [4, 6] is to let the server gradually disseminate game states in an *on-demand* way. For example, a server could gradually reveal the map information as a player explores it. Unfortunately, this state exposure control introduces delay to the client, which could disturb the player's perception of the simulated game environment. For example, if required state (e.g. map information) is delayed, the client side would have to stall the object from moving in the display. Furthermore, it is also commonly believed that on-demand loading causes more bandwidth consumption and a higher CPU load. Both directly link to a game server's scalability. Because of these two concerns, most existing online games choose *not* to do on-demand loading of game states but rather use *eager-loading*. With eager-loading, the static game states are loaded when or even before the client joins the game, and dynamic state updates are broadcasted to all the clients regardless of whether they are needed or not. However, eager loading means easy cheating.

In this paper, we examine the resource requirements of various state information dissemination strategies and study the feasibility of using on-demand dissemination strategies in online games. In particular, we consider three strategies: 1) eager loading – load all the information to the client whenever it is available to the server; 2) on-demand loading – load a state to a client upon requests; and 3) on-demand preloading – load game states that are needed and the ones that are going to be requested by clients.

We evaluate these strategies based on four perspectives: 1) the amount of excess information loaded to the client, 2) the delay incurred to the player, 3) the bandwidth consumption at the game server, and 4) the server CPU load.

We believe that it is worthwhile to evaluate these dissemination strategies for two reasons. First, although online games are interactive applications, they have been shown to have a higher delay tolerance than normal interactive video and audio. Sheldon et.al. [7] recently showed that high latency (up to 200 msec) does not affect the play of a popular Real Time Strategy (RTS) game, WarCraft, and has negli-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

NOSSDAV'04, June 18–18, 2004, Cork, Ireland.

Copyright 2004 ACM 1-58113-801-6/04/0006 ...\$5.00.

gible effect on the outcome of the game. Second, although a pure on-demand strategy could introduce considerable delays, an on-demand strategy with preloading could keep the client play smooth by predicting and preloading information that will be needed by the client. The current online games' strategy of loading all static information in the beginning as well as eagerly broadcasting game states to all users is just one extreme case on the spectrum of all preloading strategies. Given that game-cheating is a major threat, we believe it is worthwhile to consider the tradeoff between limiting the state exposure to the client and the delay introduced to the users.

Our study is conducted on a multiplayer game called OpenGladiator [8], which is a WarCraft style RTS game but is simpler in many respects and is open source. OpenGladiator was developed by programmers who went on to develop game software for Blizzard Entertainment.

Many recent works have studied various cheat prevention techniques and protocols [4, 5, 9]; however, there are far fewer works describing the system resource requirements and the performance impacts to users for the proposed anti-cheating techniques. Pellegrino et.al. [10] studied the bandwidth requirements for state consistency in various game architectures, but that work purely focused on the state consistency rather than state over-exposure at the client side. Mauve et. al. proposed a proxy system architecture to control the state dissemination but did not measure the system resources used. Similar control state dissemination has been studied in the context of multicast [11, 12] but not for cheat prevention for online games. In our work, we also consider other strategies such as eager loading with encrypted states and sending decryption keys in an on-demand way to save bandwidth and avoid state loading delay. It turns out, however, that on-demand loading of decryption keys does not reduce delay in the game we studied because most of the map information and state updates are small and have a similar size to the decryption keys.

The rest of the paper is organized as follows: Section 2 describes the model and assumptions we made about the online games and fully describes the three state dissemination strategies. Section 3 describes our evaluation approach and measurement results. Section 4 concludes the paper and discusses some future work.

## 2. STATE DISSEMINATION STRATEGIES

In this study, we only consider the client-server architecture. Although other architectures [13, 10, 6] exist, the client-server structure is still the most widely used because of its simplicity and easy subscription control.

We consider state dissemination within a single *game session*, which starts with clients joining the game server and ends when some group of players wins the game. Within a game session, the client software executes a main event loop, which includes reading input from the local player, computing local states, sending and receiving updates from the server, and rendering the local display. At the server side, a game server handles client join requests at the beginning of a game session. Once a group of players joins the session, the server needs to receive state updates from clients, control state consistency, and forward updates to clients.

### 2.1 Game States, Vision and Knowledge Sets

In a client-server architecture, the game server maintains consistency control. Our work focuses on state dissemination strategies, and to simplify the discussion, we assume states at all clients and the server are consistent due to the



Figure 1: Game Snapshot

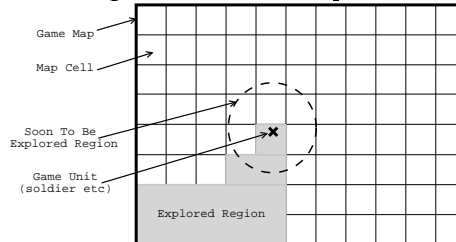


Figure 2: Explored Area and Client's Vision

consistency control at the game server. Although state inconsistency during a short period is unavoidable in online games due to the delay between client and server, many previous works have addressed this issue [9, 10]. In reality, all the state dissemination strategies discussed in this paper must be combined with a consistency control mechanism in order to be safely deployed for online games.

We further assume the game server has all of the states of the game. These states include both static information that is unchanged throughout a game session (e.g. a map) and dynamic information that changes within a game session (e.g. a soldier's position).

A client does not need to know all the game states in the server in order to play. Using the game map as an example, we call the part of map that is visible to a player its *vision*, which is the explored portion of the map. The client can render the display well as long as all the map information for its vision is available at the client side. The client's vision determines the minimal set of states (called its *vision set*) that is relevant to local play at the client side. These states include the map information for the vision, and all of the objects that are visible to the client.

Figure 1 shows a snapshot of the OpenGladiator game in which a soldier is exploring the map. Figure 2 illustrates the basic concept of a client's vision and how the vision expands in the playing process. In Figure 2, an abstracted game map is divided into many cells. Each cell could be one of many predefined geographic states, such as sand, grass, woods, etc. The cell can be in any shape; however, in this paper, we only consider 2-D square cells. A player's current vision (the gray region in the map) is the region that has been explored by the player. The vision expands when an object controlled by the player moves to a new cell. This is true in most of RTS games, where part of the game is to explore the geographic information. In first person shooting (FPS) games, client side only needs to know all the state information around the current sphere of impact. Therefore the vision doesn't require previous exploration information.

A client's vision set changes during a game session. Two factors, unit velocity and the number of units, limit the expansion-speed of a player's vision. The static map in-

formation grows as the client's vision expands. And thus static information, such as map cells, are added into the vision set. New dynamic object states are added as objects show up or removed as objects disappear from the vision.

Both static and dynamic states can be loaded to a client before they belong to its vision set. Preloading this information is unnecessary for the client to play but useful for performance concerns. We name the set of all the states known to a client its *knowledge set*. By nature, a client's vision set is a subset of its knowledge set, which itself is a subset of the server's game state set. The goal of a state dissemination strategy is then to make the knowledge set as close to the vision set as possible without causing significant performance degradation.

## 2.2 State Dissemination

Given this game model, now we consider a few state dissemination strategies. In this paper, we will not discuss how to determine the minimal state set for each player, which is game dependent. Instead, we study the system aspects of different state dissemination strategies. The comparison of these strategies is based on four aspects: the amount of excess states, the additional delay to player, network bandwidth, and CPU load.

We choose the ratio of knowledge map information versus the vision map information as the metric for measuring the excess states. Although this is not an ideal estimation of the knowledge-vision ratio, we believe it is close. We use an object map traversal time as the measurement metric for the incurred delay to the users. We measure the server bandwidth consumption through a *tcpdump* running in the network, and the CPU consumption is measured by a CPU grabber that runs at the server with the lowest scheduling priority and consumes all the free cycles that are not used by other processes.

### 2.2.1 Eager Loading

The first strategy we considered is eager loading. In this strategy, static game states are loaded into the client side at the start of a game session; dynamic states are broadcast to all clients once the server gets their updates. The client side software tracks the client's vision and decides where to display fog or map.

With the eager loading strategy, the knowledge set includes all the states in the server, and the knowledge-vision ratio is large in the beginning of a game session. Within a game session, the ratio gradually reduces as more states in the knowledge set becomes members of the vision set. In a Warcraft style game, all of the objects and all of the map could be visible to all of the players when the session gets close to the end. Under this condition, the knowledge-vision ratio is 1.

By eagerly loading map information and all object updates, the client side delay is minimal. If we consider the time for a player's unit, whose velocity is  $v$ , to move over a distance  $D$ , the traversal time is simply  $D/v$ , as all the information is available at the client.

The server consumes some bandwidth to do the eager loading (for maps, etc.) in the beginning of a game session. All of the bandwidth consumption during the game is for state update. Since all state updates are sent to all the other clients regardless of whether the object is in the client's vision or not, the server's outbound traffic will be  $(P - 1) \times S$  per update, where  $P$  is the number of players, and  $S$  is the average update message size.

Because this strategy does not take to track each client's vision, the server CPU consumption is limited to the work

for consistency control and receiving and sending state updates.

### 2.2.2 On-Demand Loading

The second considered strategy is on-demand loading. In this strategy, the server only loads the vision set to a client. Whenever the client's vision expands, the client sends a request to the server, and the server then responds with the map cells and objects' states if there are any objects on that cell. The server will track a client's vision, and only when an update is related to an object within a client's vision will the server forward the update.

With this strategy, the client has zero excess knowledge, and the knowledge-vision ratio is always 1. However, this state dissemination control introduces additional delay to the client. Every time a client's vision expands, it has to send a request. The game can only move on when the reply comes back. This delay could affect the display, and if it is too large, it becomes noticeable and annoying to players. With the traversal time as a metric, the time for a unit to move a distance  $D$  becomes  $D/v + D/C \times RTT$ , where  $C$  is the map cell size, and  $RTT$  is the round-trip-time between client and server.

Compared to eager loading, the server's inbound traffic consumption increases if explicit requests are used. However, client vision expands only when one of the client's objects move beyond its previous vision. Therefore, a state request can always be piggybacked on the object movement updates and thus requires no additional inbound traffic consumption for the game server.

The outbound traffic now has additional state reply messages but has less state updates because unrequested updates are not forwarded to clients. If we assume all objects are evenly distributed on the map, we then estimate the outbound traffic as  $(P - 1) \times S \times (M_{vision}/M_{all})$  per update, in which  $M_{vision}$  is the average vision map size of clients, and  $M_{all}$  is the whole map size. This decrement is more obvious in the beginning of a game session when clients have small visions. Once vision gets large, it gets close to the eager loading case. Our experiences indicate that with a RTS game, when all the game states are known to some clients, the game is close to ending. Most of the game session's time is spent on the exploring stage in which most clients' visions are much smaller than the whole map. Although we have no statistics to support this claim, we believe this is because most RTS games are competitions of exploring and preventing others from exploring.

With on-demand loading, the game server must do additional work to track the client's vision and filter the updates based on each client's vision. If a two dimensional array is used to represent the vision (as used in Warcraft) for each client, then checking whether an object resides in a client's vision requires only an array access. The major CPU overhead comes when the vision set is updated. When a client's vision expands, not only the new map cell information needs to be added to the client's vision set, but also all the objects' states whose positions are now in the vision. This would require the server to maintain a list of objects for each cell in the map. When an object moves across the boundary of two cells, the server would need two list operations: a detach from the list of the previous cell and an attach to the list of the new cell. Details of the CPU load measurement and discussions are presented in Section 3.4.

### 2.2.3 On-demand Preloading

The third strategy is preloading. Here, the server predicts the vision expansion of each client and sends the cells

and the associated objects (if any) before a client requests them. For example, with the simplest approach, the server predicts that the client needs all the cells around a unit's current location. The server can achieve a better prediction in a way similar to dead reckoning [14] to predict an object's future position. By preloading information for the client, this strategy helps the client reduce the delay introduced by on-demand loading. The traversal delay at the client depends on the amount of preloaded cells. If the server preloads all cells that are within a distance  $k$  (called preload depth) to the client, the client will get excess map cells quadratically to depth  $k$ , but the traversal delay will reduce to  $D/v + D/C \times (RTT - k \times C/v)$ . To fully prevent any on-demand loading delay, the preloading depth  $k$  needs to be at least  $RTT \times v/C$ .

If a game server can predict the client's requirements on time, then there will be no additional requests added to the server's inbound traffic. However, unnecessary state information is sent to client, which increases the server outbound bandwidth consumption. When an object update arrives at the server, it must forward the update to the client if the client might see it soon. We estimate the outbound traffic as  $(P - 1) \times S \times M_{knowledge}/M_{vision}$  per update, in which  $M_{knowledge}$  is the map information known by the client, and  $M_{vision}$  is the map information that is required by the client's vision.

Similarly, the only CPU load difference between this preloading mode and pure on-demand strategy is that the vision size is larger in the preloading case. The server load is similar to the on-demand case except that additional prediction is required.

### 3. EVALUATION

The experimental results reported in this paper were measured on OpenGladiator, a WarCraft like, open-source, 2D multiplayer, real-time strategy game [8]. It is a port of an old game called Gladiator that can support up to 4 players to play on one machine (with split windows). We extended it to a network based game.

An ideal platform to evaluate state dissemination strategies is a popular RTS game, such as WarCraft or Red Alert. However, this study requires modifications to the game, and none of these games' code is released to the public. A first person shooting game, Quake, is in open source and shares a similar aspect on state disseminations. However, Quake is a 3D based game, and it is complicated to identify the map information that is supposed to be visible by the player. Although there is no fundamental limit to apply the study to a 3D game, in this paper, we choose to study the state dissemination strategies in a 2D game for its simplicity.

In the OpenGladiator game, the client side keeps a 15 frame/second refresh rate. The client side update period is set to this refresh period. At every refreshing interval, the client software checks for local state changes and server updates and reports updates (if any) to the server during the frame refresh time.

Games were played between 4 players. We customized the game so that it had an automatic input from the system instead of real users. Instead of fighting with each other, we set each player to traverse an edge of the map. The map used by the game is divided into 90000 cells (300x300). The map and the game script used in the experiments are available online [15].

Our experiments were performed on 2.4 GHz Pentium-4 based PCs running Redhat Linux 9.0. The server and the clients are connected through a linux router that is also running Redhat Linux 9.0. We run Nistnet [16] in the Linux

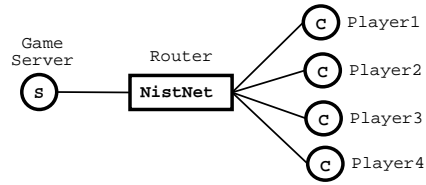


Figure 3: Experiment Topology

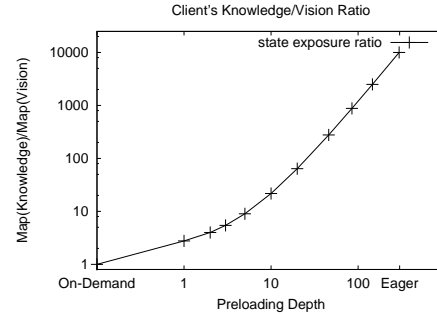


Figure 4: State Exposure Ratio vs Preloading Depth

router to emulate delay in the network. Figure 3 illustrates the experiment topology. We measure the map information exposure ratio, the map traversal time, the server bandwidth consumption, and the server CPU load. We discuss each of these in turn.

#### 3.1 State Exposure Ratio

We first measure the state exposure ratio under different preloading depths. To do this, we construct a special game scenario, in which each client has only one soldier, who starts from a corner of the map with nine cells in the client's initial vision. The soldier is programmed to explore the map automatically without requiring human input. The game server preloads map cell information for all the cells around a soldier's current position that have not been loaded before. The amount of preloaded map information is controlled by a preload depth parameter.

By varying the preload depth, we can cover a range of state dissemination strategies, including the pure on-demand one (depth=0) and eager loading strategy (depth=300, the map width). The client software logs the largest knowledge map and vision map ratio, which is used as the metric for the state exposure. We chose the highest ratio because it represents the worst possible state exposure.

Figure 4 shows the result of the maximum state exposure ratio for a range of preloading depths. This result indicates that the excess map information increases quadratically as the preload depth increases. This is because we are preloading all the cells around the soldier. A better prediction of the unit's movement could reduce the excess map information sent to clients.

The depth that we can choose to do preloading is limited by the map granularity. If a game chooses to use a large cell size and thus a small total number of cells, then the range of possible preloading depths become small.

#### 3.2 Delay Incurred to Game

In this section we study the delay incurred by the play-

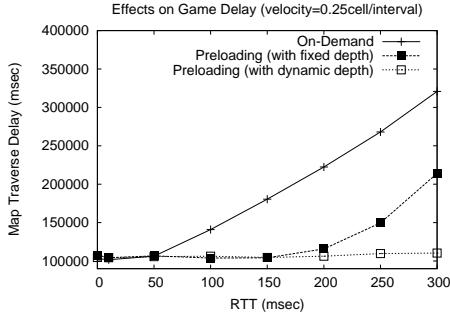


Figure 5: Traversal Delay (low obj velocity)

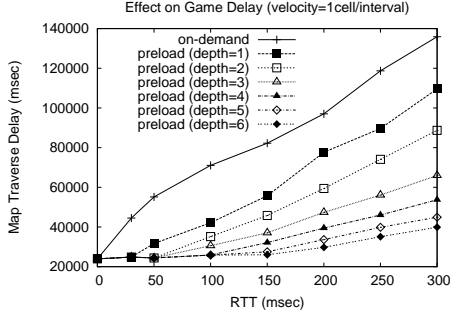


Figure 6: Traversal Delay (high obj velocity)

ers by measuring the map traversal time with various state dissemination strategies. We use the same game scenario as in the previous subsection and measure the time for a soldier to traverse an edge of the map. The state dissemination strategies considered in this experiment include: pure on-demand, on-demand with a fixed 1 cell preloading depth, and on-demand with a dynamic preloading depth that is set to  $RTT \cdot v/C$ . Here  $v/C$  is an object's movement speed in terms of number of cells per unit time.

We conduct two sets of experiments. The first one sets the maximum velocity of a soldier to 0.25 cell per refresh interval, and the second one uses 1 cell per refresh interval. Within each set of experiments, we repeat the measurement under different network round-trip-times, varying from 1ms to 300ms.

Figure 5 shows the average traversal time for a player under each RTT. For all the strategies, the traversal time does not change much when the RTT is under 50 msec. This is because the traversal time is mostly limited by the object movement speed, and the network is fast enough to get the state ready even in a pure on-demand strategy. However, with the pure on-demand strategy, as the RTT exceeds 50 msec, the traversal time increases proportionally as RTT increases.

With a preload depth 1, the traversal time is unaffected until RTT is above 150msec because preloading with a depth=1 is enough to cover small RTTs. When RTT increases, an object can move more than 1 cell away during one RTT, and thus traversal time increases even when one cell is preloaded. With dynamic preloading, the traversal time behaves as in the eager loading case, which indicates that the traversal time is unaffected even though not all the map information is preloaded. In this experiment, the maximum depth value used by the dynamic depth adaptation is 2.

Figure 6 illustrates the case when a higher velocity (1cell

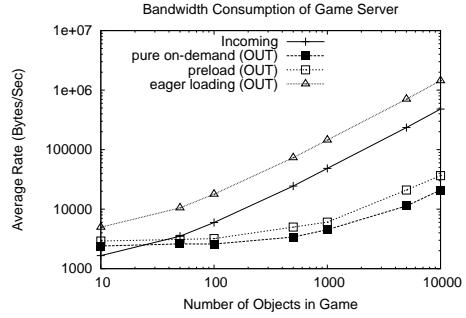


Figure 7: Server Bandwidth Consumptions

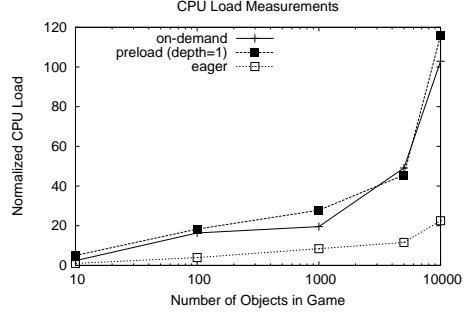


Figure 8: Server CPU Overhead

per refresh interval) is assigned to the soldier. With a high velocity, the traversal time over the same distance decreases in general because the object moves faster than the previous experiment. Because of this, deeper preloading depths are required to cover the same RTT. For example, preloading with a depth 1 would start introducing additional traversal delay when RTT is larger than 30 msec rather than 150 msec as in the previous experiments. Increasing the preloading depth helps to cover the delay by exposing more information. This is indicated by the Figure 6, which shows the traverse delay for various preloading strategies with a depth up to 6.

### 3.3 Server Bandwidth Consumption

We study the server's resource requirements in this section. In order to test the scalability for a game server under different state dissemination strategies, we measure the server traffic consumption for various state dissemination strategies when the number of objects in a game increases.

The game OpenGladiator has a small, limited number (16) of objects per client. In order to test the scalability of the game server, we modified the game to insert more objects from client side, in addition to the soldier that moves along the edge of the map. The objects we added make random moves in the game. The traffic is measured by a tcpdump running at the Nistnet machine. The measurement is made over the time that a client finishes exploring an edge of the map, and the average rate consumption is presented.

Figure 7 shows the incoming and outgoing traffic rate for a game server. In this measurement, no matter which state dissemination strategy is in use, the incoming traffic to the game server increases linearly as the number of objects increases. This proportional relationship is because every object's move would cause the client to send an update to the server. We only present the incoming traffic of the

on-demand preloading case. The results for other strategies are similar to this case. The eager loading strategy consumes more bandwidth than other strategies for outgoing traffic because updates are forwarded to all the other clients regardless of whether they are required or not. For this reason, the game server's outgoing bandwidth consumption is always higher than the incoming one when eager loading is used. The outgoing traffic consumption of the pure on-demand strategy is the lowest because the server only forwards updates that are in the client's vision. With preloading, additional updates are forwarded to a client if any objects are close to its vision. In addition, our results indicate that as the number of objects increases, the game server's outgoing bandwidth consumption becomes less than the incoming traffic because many received updates are not forwarded out.

This result indicates that on-demand loading and preloading reduces the unnecessary updates forwarded to the clients. This saves bandwidth for the game server, which reduces the cost of running a game server.

### 3.4 Server CPU load

The benefit of preloading and on-demand loading comes at a cost of additional CPU processing to track the client's vision and to check whether an update should be forwarded to a client based on the vision. We measure the average CPU load of the game server for various state dissemination strategies under the same conditions presented in previous subsection. We normalize all CPU usage to the base, which is the the CPU consumption of the eager loading with 10 objects. The normalized results are presented in Figure 8.

The measurement results show that the overall CPU usage of on-demand loading and preloading are higher than eager loading. This shows that the benefit of cheating prevention and traffic reduction comes with a cost of higher CPU consumption.

Originally, we thought that the reduction of network traffic would reduce the server's CPU consumption spent on sending and receiving packets, which might reduce the overall CPU consumption. However, our measurement shows that the CPU load of on-demand and preloading strategies are about an order of magnitude higher than the CPU load with the eager loading when the number of objects changes from 10 to 10000, and the difference increases as the number of objects increases.

We are in the process of profiling the server's CPU consumption, and in the future, we plan to break down the CPU consumption in more detail (e.g. peak CPU usage). The measurement results shown in this figure are taken in a session that is spent solely on exploring. In the future, we also plan to study the overall system resource requirements in game sessions that include real combat.

## 4. CONCLUSION

In this paper, we studied various state dissemination strategies for online games, comparing them based on excess state exposure, user delay perception, and system requirements.

Our measurement results show that on-demand loading can reduce the excess state exposure to the client and thus reduce the threat of map-hacking style game cheating. On-demand loading introduces delay to game players, but the delay can be reduced with preloading. The current state dissemination strategy is just one extreme example of preloading, and it gives out too much excess state to the clients. To have effective preloading to reduce delay, the preloading depth is determined by the round-trip-time, the game unit velocity, and the map cell size.

Our measurement results show that, with on-demand preloading, the game server's CPU consumptions increases. However, the on-demand preloading significantly reduces the outgoing bandwidth consumption from the server.

Certainly, with on-demand loading, it is possible that hackers can launch a new type of cheating mechanism – making requests to the game server for game state information to which they should not have access. Our results show that the on-demand request speed is closely associated with the unit exploration rate, which is a game design factor. By knowing the highest moving velocity of all game objects, the server can detect if the state requests may be generated by a client side hack.

## 5. REFERENCES

- [1] Erwin Lemuel G. Oliva. 131m online gamers in 2006. In *Infotech News Article*, October 2003.
- [2] Steven B.Davis. Why cheating matters: Cheating, game security, and the future of global on-line game business. In *In Proceedings of the 2001 Game Developers Conference*, 2001.
- [3] J. Yan. Security design in online games. In *In Proc. of the 19th Annual Computer Security Applications Conference (ACSAC'03)*, IEEE Computer Society, December 2003.
- [4] Matt Prichard. How to hurt the hackers: The scoop on the internet cheating and how you can combat it. In *Game Developer Magazine*, pages 28–30, June 2002.
- [5] Counter Hack. Warcraft 3 cheats and hacks. <http://www.counter-hack.net/content.php?page=warcraft3>, 2004.
- [6] Martin Mauve and Stefan Fischer. A generic proxy system for networked computer games, 2002.
- [7] Nathan Sheldon, Eric Girard, Seth Borg, Mark Claypool, and Emmanuel Agu. The effect of latency on user performance in warcraft 3. In *In proceedings of ACM NetGames 2003*, May 2003.
- [8] SnowStorm Entertainment. Opengladiator multiplayer game, 2003.
- [9] Nathaniel E. Baughman and Brian Neil Levine. Cheat-proof payout for centralized and distributed online games. In *INFOCOM*, pages 104–113, 2001.
- [10] J.Pellegrino and C.Dovrolis. Bandwidth requirement and state consistency in three multiplayer game architectures. In *In proceedings of ACM NetGames 2003*, May 2003.
- [11] Brian Neil Levine, Jon Crowcroft, Christophe Diot, J. J. Garcia-Luna-Aceves, and James F. Kurose. Consideration of receiver interest for IP multicast delivery. In *INFOCOM (2)*, pages 470–479, 2000.
- [12] L. ZOU, M. AMMAR, and C. DIOT. An evaluation of grouping techniques for state dissemination in networked multi-user games, August 2001.
- [13] C. Diot and L. Gautier. A distributed architecture for multiplayer interactive applications on the internet. In *IEEE Networks magazine*, 1999.
- [14] Martin Mauve. How to keep a dead man from shooting. In *Interactive Distributed Multimedia Systems and Telecommunication Services*, pages 199–204, 2000.
- [15] Shanshan Ding. Opengladiator scripts and results. <http://www.arches.uga.edu/~shanshan/opengladiator.html>.
- [16] Nistnet network emulator. <http://snad.ncsl.nist.gov/itg/nistnet/>, 2000.