Strifeshadow Fantasy: A Massive Multi-Player Online Game

Hang T. Chan
Unrealize Development Team, BMK Networks
Unit 2401A, Part-in Comm. Ctr.
56 Dundas St., Mongkok, Kln., Hong Kong
Email: xunreality@hotmail.com

Rocky K. C. Chang
Department of Computing
The Hong Kong Polytechnic University
Hung Hom, Kowloon, Hong Kong
Email: csrchang@comp.polyu.edu.hk

Abstract— **Strifeshadow Fantasy (SSF) is a massive, multi-player online, role-playing game. Players of this game, acting as avatars, search for the ancient signs, and their goals are to defeat the God of Destruction. Players can adventure in the game alone or co-operate with the others through the chat box. SSF is available for free and there are currently more than 10,000 registered users. In this article, we highlight the overall software architecture of SSF, which is based on a simple server-client model and HTTP. We will also describe in details two problems encountered in the course of designing SSF and the solutions to them. The first one is a local state consistency problem which is to ensure that each client participating in the game will eventually receive all the state updates once and only once. The second one is a connection jamming problem that is a result of using nonpersistent HTTP connections for the communication between the game server and clients.**

I. Introduction

Massive multi-player online game (MMOG) is a rapidly growing area in the video game industry. Unlike the earlier networked games, MMOG aims to support a very large number of concurrent users, in terms of thousands and even tens of thousands. Moreover, the MMOGs are usually role-playing style games in that players could continuously involve in a story as long as they want, e.g., Ultima Online and EverQuest. While there are many areas of technical expertise go into the development of MMOG, we only consider the network-related issues here, such as bandwidth scalability, state consistency, fairness, accessibility, player-collaboration, and security.

Since the data exchanged between a game server and the clients in a typical server-client game architecture grows exponentially with the number of users, the server's network bandwidth is often the bottleneck. Therefore, one important design issue is the dimensioning of the network bandwidth. On the other hand, others have proposed peer-to-peer [1] and mirror-server game architectures to increase the network bandwidth. With a non-centralized server model, state synchronization, however, becomes an important issue, and various mechanisms have been proposed to address it, e.g., [2], [3].

In a centralized server model where all the game data resides, players request the game states asynchronously. Even if the server pushes the states at the same time to all players, the players still receive the states in different times because of the different latency and packet loss experienced in different parts of the Internet. Therefore, some players may receive, and therefore react to, the current state of the game much faster than the others. We call this problem a *global state consistency problem* which obviously has implication on the fairness of playing the game. Kim et al proposed a global timestamp-based approach to tackle this problem [4]. Moreover, there is a variant of the state consistency problem that ensures that each player *eventually* receives the same state update once and only once. It is clear that this *local state consistency problem* does not concern the exact times of receiving the game states.

Accessibility concerns whether a player can assess the game of interest anywhere and anytime he wants, e.g., through the ubiquitous Web interface. Or the player is required to pre-install some program in his machine.

Since the main feature of an MMOG is for players to interact with each other, an MMOG must provide facilities for players to communicate and collaborate with other players throughput the game session. Moreover, the communication should be secure, so that other rival groups of players would not be able to listen to their private messages. Therefore, data security is also important in this context and in other forms of cheating scenarios [5].

In this paper we describe how the design of Strifeshadow Fantasy (SSF) addresses some of the issues above. SSF is based on a server-client model, and the game server and clients communicate using HTTP. Therefore, players can assess the game anywhere and anytime. Even with a centralized server, as will show later, there are effective ways to scale to the number of players. Moreover, we will show how SSF solves the local state consistency problem.

The rest of the paper is organized as follows. In section II, we will first highlight SSF's architecture and the connection model. In section III, we consider the local state consistency problem and explain why a simple action-flag approach would not work. Subsequently we introduce a action-log table approach to solve the problem. In section IV, we consider a connection jamming problem and examine three possible solutions. In section V, we highlight other problems that we have encountered throughout the development of SSF. Finally we conclude this paper with some future work.

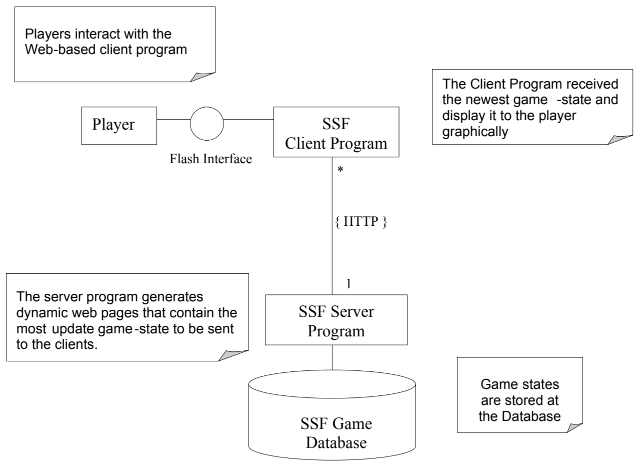
Players interact with the Web-based client program
Player
Flash Interface
SSF Client Program
The Client Program received the newest game -state and display it to the player graphically
*
{ HTTP }
1
The server program generates dynamic web pages that contain the most update game-state to be sent to the clients.
SSF Server Program
SSF Game Database
Game states are stored at the Database

Fig. 1. SSF's architecture.

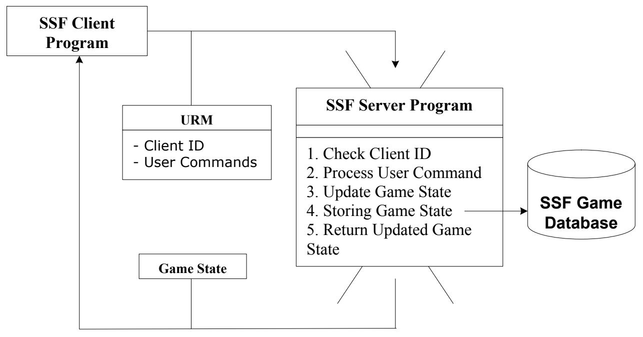

Fig. 2. Operational diagram of the SSF server.

II. STRIFESHADOW FANTASY (SSF)

SSF is a massive multi-player online role-playing game which was 100% developed locally in Hong Kong [6]. The first author single-handedly developed and maintained the entire system on part-time basis. SSF has over 10,000 registered players who come from various countries. SSF's story is about a fantasy world in which there is a conflict between the God of Construction and God of Destruction. Players, acting as avatars, search for the ancient signs, and their ultimate goals are to defeat the God of Destruction.

A. SSF's architecture

SSF is based on a very simple server-client model. As shown in Fig. 1, there is a central server hosting the SSF server program or game engine, and a SSF game database. Players connect to the game server through the SSF client application in order to participate in the game. Although the server-client model may suffer from the bandwidth scalability and single-point-of-failure problems, it is perhaps the best model to start with for the development of any MMOG. It can also be adapted to the mirrored-server model in the later stage of the development.

Besides the ease of implementation, there are a number of issues that can be easily addressed by a server-client model. The first one is system robustness. Since the game server serves each participant independently, the failure of any of the clients has no effect on others. This obviously cannot be said for the peer-to-peer model.

Another is data security requirement. The SSF game data is protected from hacking or other abuses by splitting the entire application into server and client parts. The server application part handles all the computation and processing of the game data. The client application part, on the other hand, is responsible for retrieving the most updated game data and displaying them. Therefore, the game data is in the "read only mode" on the client side. If the client needs to modify any of the game data, it has to send a valid procedure call to the server to request the modification.

Moreover, the ability of recovering data in an MMOG is an important requirement, because it is quite common for data lost during the message transmission from the server to client or vice versa. In SSF, data recovery is ensured that even if there is some game data missing or corrupted during the transmission. The client side can still reconstruct the full game state in the next successful data transmission.

B. SSF's connection model

SSF's connection model is similar to that used for the Web chat rooms, where the clients continue to send HTTP requests to the chat-room server to refresh the Web page and receive the most updated chat history. In SSF, each client sends a Update Request Message (URM) using HTTP to the server at regular intervals called *game rounds*. Each URM consists of two parts. The first part contains a client ID, and the second contains the user commands.

C. SSF server's design

The SSF server is written using Active Server Page (ASP) that can create powerful Web-based applications through the combination use of HTML, scripts, and Microsoft ActiveX server components. It handles the following tasks for the game:

1) Handle the member registration, login, logout, and game saving operations.
2) Receive and validate the URMs by checking its client ID, and then process the user command that is contained in the URMs.
3) Handle any modifications and updates of the game state.
4) Write the most updated game states to the SSF Game Database.
5) Send back the most updated game state to the clients in the format of HTTP Query String.

D. SSF client's design

The SSF client provides the SSF virtual environment interface for users to interact with, and it also acts as a communication bridge between the current user and the SSF server. It is written using Macromedia Flash 4.0. The client handles the following tasks for the game:

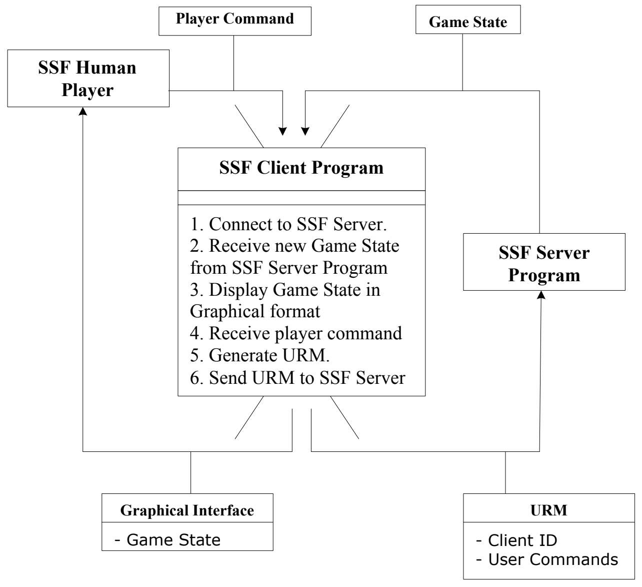
Player Command
Game State
SSF Human Player
SSF Client Program
1. Connect to SSF Server.
2. Receive new Game State from SSF Server Program
3. Display Game State in Graphical format
4. Receive player command
5. Generate URM.
6. Send URM to SSF Server
SSF Server Program
Graphical Interface
- Game State
URM
- Client ID
- User Commands

Fig. 3. Operational diagram of the SSF client.

1) Provide the interactive GUI that represents the SSF virtual world environment.
2) Receive user commands from the GUI and convert them into HTTP Query String format
3) Build the URMs and send them to the server at regular intervals.
4) Receive game states from the server, and display them by converting the game states into meaningful graphics and animations,

Fig. 4 shows the states that a typical SSF client would go through.

E. The SSF game database

The SSF game database stores member accounts, game states, and game activities. It is responsible for providing the correct game states to the SSF server. The database is based on the relational database technology that is quite suitable for storing game data, because it require few assumptions about how data is related or how it is extracted from the database.

III. A LOCAL STATE CONSISTENCY PROBLEM

Sometimes the actions taken place at the SSF game world cannot be displayed correctly on the client side, which is due to the various speeds and lossy conditions of the clients' networking environments. Therefore, the design of SSF must mask these conditions, such that each action taken place in the game world should be eventually displayed on each client once and only once.

A. A action-flag approach

A simple solution would be to create an action flag to indicate the status of each game action. For example, if the action has taken place, then set the value of that action flag to *True*; otherwise, it is set to *False*. Therefore, when a client

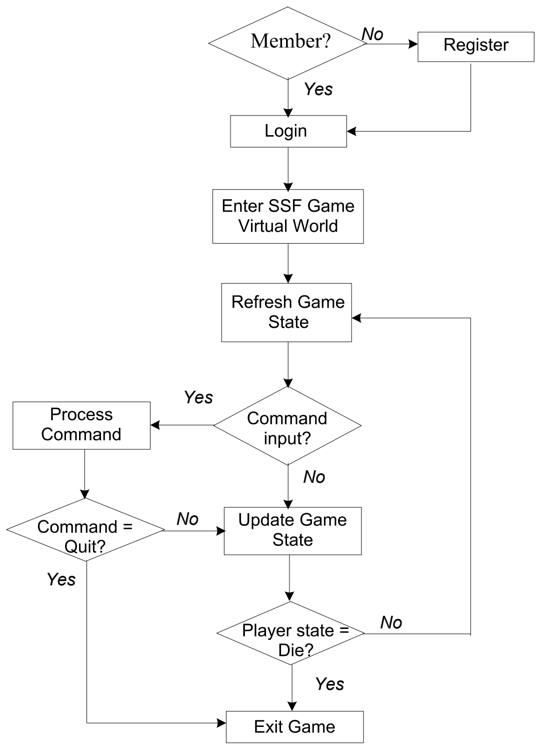

Fig. 4. The game states of a SSF client.

requests the latest game state, it would be able to know whether it should display the action or not based on the value of the action flag. After the action flag has been set to *True* for a certain period of time, the action flag will be reset back to *False* again to indicate that the action is already completed.

Unfortunately, this simple solution turns out to be a very bad design. Some undesirable results will happen when the client network connection is either too fast or too slow. The correct case is shown in Fig. 5, where the action has taken place and its action flag is set to *True*, then the client requests for the game state, and displays the action properly. After the value of the action flag set back to *False* after a period of time, the client requests the game state again and this time it doesn't display the action.

In Fig. 6, a client has a slow network connection. Therefore, the action has taken place and its action flag is set to *True*, but one of the client's network connection is so slow or lossy that the action flag has been set back to *False* before that client successfully requests the game state. As a result, the action is not displayed in that particular client.

In Fig. 7, a client has a "too fast" network connection. The action has taken place and its action flag is set to *True*, but one of the client's network connection is fast enough to request for the game state twice before the action flag is reset back to *False*. As a result, the action is displayed twice in that particular client.

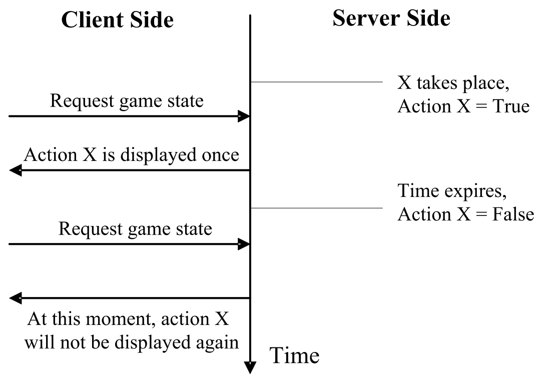

Fig. 5. Action X is displayed correctly using the action-flag approach.

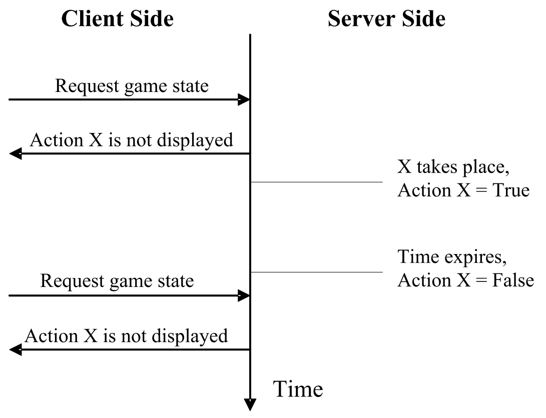

Fig. 6. Action X is displayed correctly using the action-flag approach.

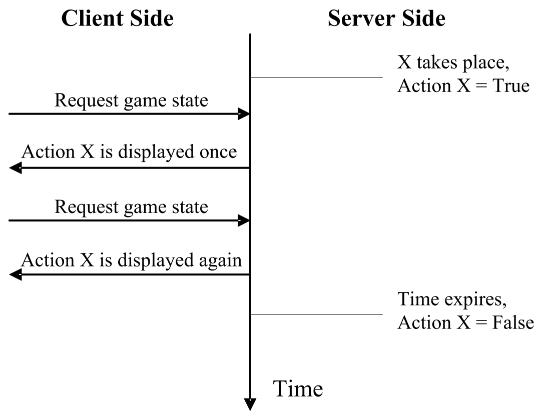

Fig. 7. Action X is displayed correctly using the action-flag approach.

Client	Pointer
A	102
B	103

Action Log Pointer

ID	Action
101	B attacks C
102	C Heals C
103	D talks to A

Action Log Table

Fig. 8. A action-log table and a action-log pointer for a client.

B. A action-log table approach

Another approach is to create a action-log table in the database. Whenever an action has taken place, the SSF server will insert a new record into the action-log table, and that record will contain all the information related to that action. In addition, the SSF server will assign an action-log pointer to each of the clients, which points to the last action record that the client has retrieved. An example is shown in Fig. 8.

When a client requests the SSF server for the latest game state, the SSF server will retrieve those unread action records by looking up the value of that client's action log pointer. After sending the state updates, the SSF server updates the client's action log pointer by making it point to the last action record that the client has retrieved. This solution works correctly, because the action retrieval model is independent of the network connection. No matter whether the client has fast or slow network connection speed, the sequence of actions are still displayed correctly on the client side.

IV. A CONNECTION JAMMING PROBLEM

As the connection model between the SSF server and clients is based on HTTP, the server cannot initiate a connection to the client and send the latest game state to them. Therefore, the SSF client has to frequently query the SSF server for the latest game state. As a result, setting a proper request rate is a key issue to avoid a connection jam which could seriously affect the performance of the entire game system. In the following, we describe and evaluate three approaches to this problem.

A. A periodic-updating approach

An obvious approach is to send URMs to the SSF server at fixed intervals (e.g., 500ms). Even if there is no reply message received from the server, or the reply message is delayed, the client still sends the URMs to the server on schedule. If the interval is too short, unnecessary URMs will be sent out, as shown in Fig. 9.

If the SSF server is already congested with a lot of requests, it will take a longer time for the server to reply the URM. Therefore, the server will be even more congested if each client keeps on sending URMs periodically. This is verified by the data in Fig. 10, where the clients ping the server at a regular interval of 500ms. At the beginning, the connection is quite stable, but as time goes by the network connection becomes slower and there are more and more lag spikes. After 400 game

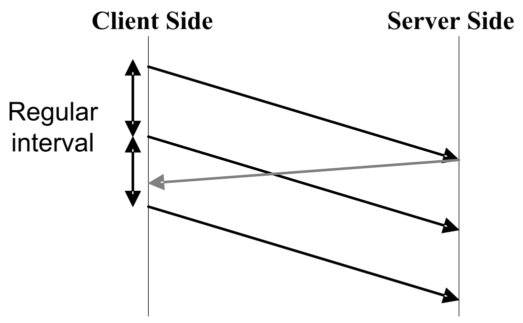

Fig. 9. A periodic-updating approach.

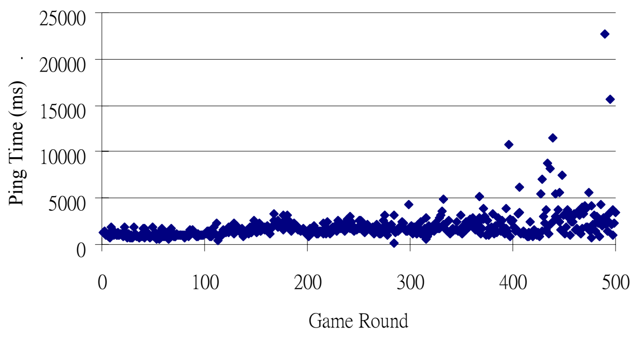

Fig. 10. Test results for the periodic-updating approach.

rounds, the network connection speed becomes unacceptably slow (the average ping time reaching 4000ms).

B. A client-side timeout approach

A second approach attempts to improve the first one by keeping more states about the requests. In this approach, a client sends out a URM if (a) it is the first URM, or (b) the timeout limit (e.g., 3 seconds) is reached, or (c) a reply message has been received. In this case, there will only be at most one URM, except that the server's reply message cannot reach the client before timeout.

This solution generally works better than the periodic-updating approach. However, the same problem could occur if the reply message cannot reach the client on time, as shown in Fig. 11. Fig. 12 shows the results for this approach and the timeout value is set to 3000ms. At the beginning, the connection is very stable. However, when there are a few pings exceeding the timeout limit, the connection starts to become unstable, and more ping exceeds the timeout limit. Again, after 400 game rounds, the network connection speed becomes unacceptably slow (the average ping time reaching 3000ms).

C. A server-side timeout approach

Unlike the second approach where clients regulates the sending rate of URMs, this approach allows the SSF server to timeout clients. In this approach, a client sends out a URM if (a) it is the first URM, or (b) a reply is received. Moreover,

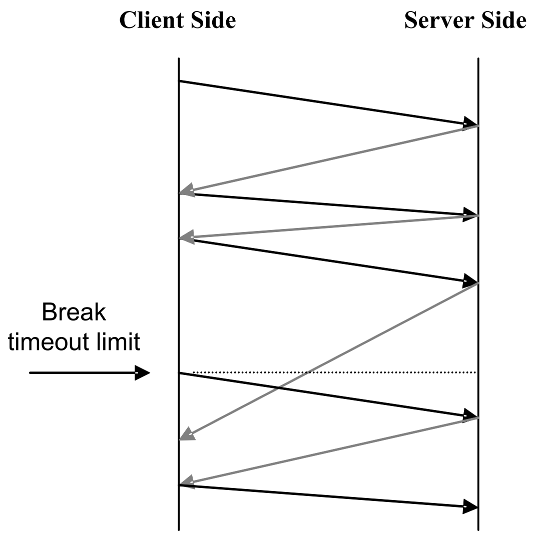

Fig. 11. A client-side timeout approach.

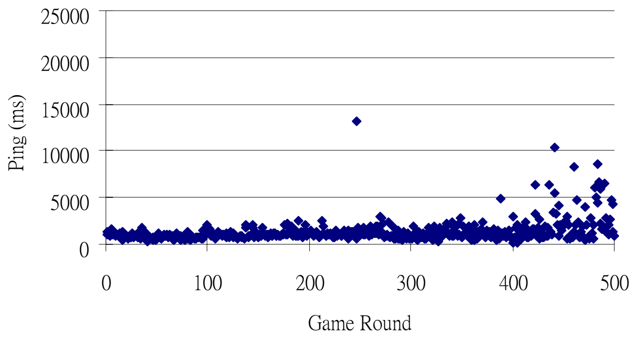

Fig. 12. Test results for the improved periodic-updating approach.

the server keeps a timer for timing the client's next URM. If the next URM is not received within the timeout period, the SSF server considers that the client is not connected, therefore removing the client from the connected client list. In this case, the client needs to register again to join the game. As shown in Fig. 13, there is at most one URM sent from a client at any time. The data in Fig. 14 shows that the connections using this approach are very stable from the beginning to the end.

V. Other Issues

Besides the problems described above, we have encountered a number of other problems in the design and implementation of SSF. We briefly describe some of them in the following.

A. High server resources usage

There is a serious shortcoming of using ASP for the server-side program. Every time the client sends a URM requesting the latest game state, the server must create a new process to run the server side scripting in order to respond to it. Creating a process for every single request requires time and significant

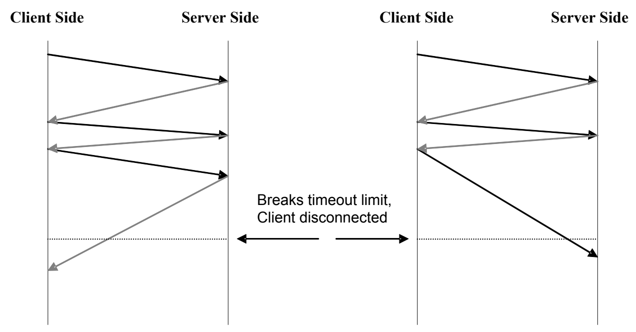

Fig. 13. A server-side timeout approach.

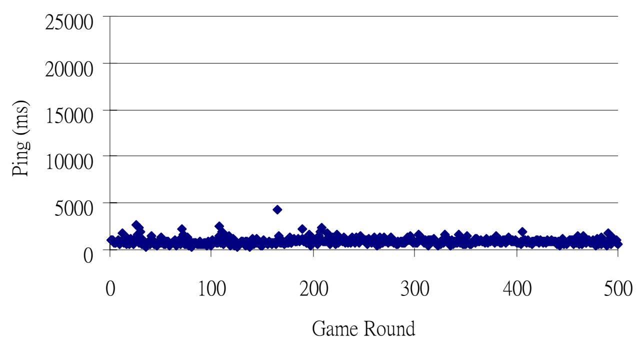

Fig. 14. Test results for the server-side timeout approach.

server resources, which limit the number of requests a server can handle concurrently.

B. Excessive database connection

Another disadvantage of using ASP is that the variables stored at a particular process will be removed from the memory when the process ends. Therefore, when the server needs to store the variables for future usage, it needs to write them into hard disk or database file. As a result, it increases the number of hard disk read/write operations, thus imposing significant workload on the server.

C. Message delay

The global state consistency remains an open problem to solve in SSF. Although the SSF could resolve the local state consistency problem using the action-log table approach, the clients are not guaranteed to receive the most updated game state at the same time. To provide global state consistency, other connection model, such as multicasting, may be a better choice.

D. Large connection overhead

Having a large overhead when there is nothing to be updated is another important issue to address, i.e., the server's replies are essentially empty. In this case, the game system should be intelligent enough to eliminate, or minimize, the protocol overhead and message processing.

VI. Conclusions and future work

Developing a highly scalable, efficient, and fair MMOG remains a very challenging task. In this paper, we have described some of the technical details about Strifeshadow Fantasy (SSF), a working MMOG that is actively used by a large number of users. Although there are quite a few MMOGs available today, we are not aware of any articles providing sufficient details of their designs and describing how they address some of the network-related issues described in this paper. The work on MiMaze has been described in details in [7]; however, it does not seem to be under active use as for the case of SSF.

As seen from this paper, there remains quite a number of issues for improving SSF's performance and scalability. One of the issues concerns the connection model. Nonpersistent HTTP connections are currently used between the server and clients. Another alternative is to use persistent HTTP connections or even sockets, so that only a single connection is required for each client. We have used sockets to implement a prototype that incurs less overheads and message processing. Another possibility is to use IP multicast or application-level multicast, so that the server can push out the game state updates to all clients at the same time.

References

[1] B. Hoyt, "Full Circle: Why Peer-to-Peer Architectures Should Replace Client/Server in MMOGs" available in http://www.gignews.com/spotlight0901_hoyt.htm.

[2] E. Cronin, et al, "An Efficient Synchronization Mechanism for Mirrored Game Architectures," *Proc. NetGames*", 2002.

[3] J. Steinman, et al, "Scalable Distributed Military Simulations Using the SPEEDES Object-Oriented Simulation Framework," *Proc. Object-Oriented Simulation Conf.*", pp. 3-23, 1998.

[4] S. Kim, F. Kuester, and K. Kim, "A Global Timestamp-Based Scalable Framework for Multi-player Online Games," *Proc. IEEE Fourth Intl. Sym. Multimedia Software Engineering*, 2002.

[5] J. Weisz, "Detecting Cheaters in a Distributed Multiplayer Game," available from http://www-2.cs.cmu.edu/afs/cs.cmu.edu/user/mjs/ftp/thesis-03/weisz.pdf.

[6] Strifeshadow Fantasy, http://ssfantasy.n3.net/.

[7] C. Diot and L. Gautier, "A Distributed Architecture for Multiplayer Interactive Applications on the Internet," *IEEE Network*, pp. 6-15, July/Aug 1999.