Selected Techniques to Improve Communication in Massively Multiplayer Games

Lucas E. Machado Bruno Feijó Lauro E. Kozovits*

VisionLab/IGames, Dept. of Informatics, PUC-Rio, Rio de Janeiro, Brazil

Abstract

There are no papers or books in the literature that analyze the most suitable techniques to improve communication in massively multiplayer games in a detailed and implementation-oriented way. Most of the results and recommendations in this area are buried in proprietary systems and classified documentation. Based on experiments in the VLab/IGame Laboratory, this paper presents selected techniques and practical recommendations for the development of Massively Multiplayer Games. Optimization techniques in the use of sockets are given. Technologies capable of handling many sockets at once are explained. Also techniques for optimizing communication in virtual environments are presented.

Keywords: Distributed Virtual Environments, Networked Virtual Environments, Massively Multiplayer Games

Authors' contact:
{lucas,bruno}@inf.puc-rio.br
*lek@3Dgames.com.br

1. Introduction

Massively Multiplayer Games (MMGs) are computer games where thousands of players simultaneously connect themselves to persistent worlds using the Internet. In this type of game, the world continues to evolve even after the player disconnects him/herself. The characters used by the players are not lost after the player exits the game, allowing the continuous evolution of the characters. In order to process the great number of MMG players in such a complex environment, optimization techniques and efficient technologies must be applied in the communication between clients and servers.

Knowledge and data about techniques and technologies for massively multiplayer games are still scarce in the literature. This paper aggregates valuable practical knowledge on distributed programming and present the best techniques to optimize communication in MMGs, in an integrated way that cannot be found elsewhere. Most of the ideas of this paper are derived from a previous work by the first author [Machado, 2005].

Some of the techniques presented on this paper are operating system specific. They were selected because they offer better performance than other operation system independent techniques.

This paper organizes the subject in three main areas: techniques using TCP, UDP, and sockets (section 3); I/O processing of multiple sockets on servers (section 4); and techniques for networked virtual environments (section 5).

2. Related Work

The technology used to fuel most of the today's massively multiplayer games is still a mystery and the literature is too scarce.

Books on MMG are beginning to appear. However, some books are more an informative reference than a source for implementation techniques [Alexander 2003]. Others books only introduce specific styles of multiplayer games [Penton, 2003]. Some game conferences give important tutorials and lectures on the subject, but they are not usually published [GDC, 2006]. Also there are good articles in developers sites (www.gamedev.net), but either they are too superficial or too narrow.

Well-known general books are mandatory references, but they do not focus on MMG. For instance, general aspects of networked virtual environment are covered by [Singhal et al., 1999]. Snader [2000] presents a good work on TCP/IP programming. An overview of peer-to-peer technology can be found in [Oram, 2001].

Architectures and protocols for distributed virtual environments adopted by IEEE and OMG are too heavy and excessively general to be applied in games [IEEE, 1993] [Kuhl, 1999].

3. Techniques Using TCP, UDP and Sockets

On its foundation, all distributed systems must deal with TCP, UDP or Sockets in some level. The efficient use of those technologies is very important because they affect how communication is performed.

3.1 The Naggle Algorithm and The Delayed Ack

If TCP is to be applied at a certain point in the communication of a distributed system, it is important that two concepts be well understood: the Naggle Algorithm and the Delayed Ack.

The Naggle Algorithm consists of not sending a message when the message is small and there is another buffered message that has not received an Ack yet. A message is small if it is smaller than the maximum segment size that can be sent to another peer.

Suppose that a server wants to send the position of two players to a client. The first position is sent with the socket function `send()`, then the second position is sent with another `send()`. In this example, the second position will only be sent after the server receives an Ack for the first message and this can take several milliseconds on a WAN. A better strategy would be the server to aggregate the positions in a single message and send them both in a single function call of `send()` or to disable the Naggle Algorithm using the following code:

```
BOOL b = true;
setsockopt(socket, IPPROTO_TCP,
TCP_NODELAY, (char*)&b, sizeof(BOOL));
```

Another important concept in TCP is the Delayed Ack. Every message received in a TCP communication is not immediately acknowledged. Instead, TCP "waits" for the receiver to respond to the sender of the message. If the receiver responds to the sender, TCP will piggyback the acknowledgement in the response. If not, after some milliseconds, TCP sends an Ack to the sender.

3.2 Decreasing Buffer Copies in WinSock

When an application receives or sends a message, a buffer copy is performed, transferring data from the application buffer to the internal socket buffer and vice versa. WinSock allows the removal of this transfer, making the communication use only the application buffer.

In order to use this capability, WinSock overlapped I/O operations must be used [Microsoft, 2005]. An overlapped operation is an operation that is asynchronous (that is, it returns immediately, but the result is sent later to the application). After the socket is created, the following functions can be used: `WSASend`, `WSASendTo`, `WSARecv`, and `WSARecvFrom`. These functions receive a variable `WSAOVERLAPPED`, which is a structure that provides a communication medium between the initiation of an overlapped I/O operation and its subsequent completion. This variable should first be set to zero. Below is an example of an overlapped send operation:

```
WSAOVERLAPPED snd_op;
ZeroMemory(&snd_op, sizeof(snd_op));
WSASend(socket, buffer, num_of_buff,
bytes_sent, flags, &snd_op, NULL);
```

A final step is necessary in order to make WinSock use only the application buffer for the communication. This is done by setting the socket buffers to zero, as shown below:

```
int zero = 0;
setsockopt(socket, SOL_SOCKET, SO_SNDBUF,
(char*)&zero, sizeof(zero));
setsockopt(socket, SOL_SOCKET,
SO_RCVBUF,  (char*)&zero, sizeof(zero));
```

3.3 Using Connected UDP Sockets

When the socket function `connect()` is called in a UDP socket, no message is sent. Instead, only the remote address and port are saved in the socket. With this information, it is possible to use the function `send()` with the socket instead of the function `sendto()`.

On some operating systems, calling `sendto()` makes the kernel do the following operations: to connect the socket used, to send the message, and to disconnect the socket. Connecting and disconnecting the socket like this can take as much as a third of the processing time for sending the message [Partridge et al., 1993].

Besides the performance improvement, connecting UDP sockets allows for asynchronous error capturing. When an UDP message is sent to another peer and no applications are receiving messages at the destination port, an ICMP port-unreachable message is returned. The only way to receive this message is with a connected UDP socket.

3.4 Sending Data from Multiple Buffers in Sockets

When a message of dynamic size needs to be transferred using the common Sockets API `send()` or `sendto()`, it is necessary that the message be composed in a single buffer in order that the transference be performed.

An example of a message of dynamic size is a chat message that has a static header part and a dynamic body part.

WinSock allows the composition of the dynamic message directly in the internal socket buffer, thus reducing memory transfers. An example that demonstrates this functionality is shown below:

Header `header;`
Fill header parameters

```
WSABUF msg_buffer[2];
msg_buffer[0].buf = &header;
msg_buffer[0].len  = sizeof(header);

msg_buffer[1].buf = chat_msg;
msg_buffer[1].len  = chat_msg_size;

WSASend(socket, msg_buffer, 2,
&bytes_sent, flags, NULL, NULL);
```

The type `WSABUF`, used for the operation, has the following format:

```
typedef struct _WSABUF {
```

```
   u_long len;
   char FAR * buf;
} WSABUF, FAR * LPWSABUF;
```

where `len` is the buffer length and `buf` is a pointer to the buffer.

The description of `WSASend` is as follows:

```
int WSAAPI WSASend(SOCKET s, LPWSABUF
buf, DWORD cnt, LPDWORD sent, DWORD
flags, LPWSAOVERLAPPED ovl,
LPWSAOVERLAPPED_COMPLETION_ROUTINE func)
```

where `s` is the socket, `buf` is a vector of `WSABUF`, `cnt` is the number of buffers in the vector, `sent` is the number of bytes sent, `flags` is similar to the flags used in the conventional send, `ovl` and `func` are used for overlapped operations.

3.5 Implementing Reliability on UDP

There are many cases were it is useful to have a reliable UDP protocol implemented. This protocol makes sure that the messages sent are received at the destination, but no order is enforced in the arrival of the messages to the application.

The usual way to implement reliability for message communication is using timers and retransmission. When a message is sent, a timer is started. If a time limit is reached, the timer will expire and the message will be retransmitted. When the other peer receives a message, it replies with an acknowledgment of arrival.

An important decision that must be handled is the choice of a proper time to wait until retransmission, called Retransmission Timeout (RTO). If a bad choice of RTO is made, a waste of time and computational resources will occur.

For a good choice of RTO it is important to take into consideration the estimated time that a message takes to be transferred from a sender to a destination and back to the sender. This time is called round trip time (RTT). Considering that there is an error between the current RTT and the estimated RTT (SRTT), the following algorithm (known as the Jacobson/Karels algorithm) can be applied:

$error = RTT - SRTT$
$SRTT = SRTT + (error / RTTINF)$
$VARRTT = VARRTT + [(|error| - VARRTT) / VARRTTINF]$
$RTO = SRTT + (VARRTT * VARINC)$

Where:

RTTINF is a value that defines how much of the difference between the current and estimated round trip time is transferred to the estimated value;

VARRTT is the deviation (variation) between different round trip times, that is: an estimative of the variation of RTT;

VARRTTINF is similar to RTTINF and defines how much of the difference between VARRTT and *error* is transferred to VARRTT;

VARINC is a value that increases the influence of VARRTT.

Suggested values for RTTINF, VARRTTINF and VARINC are 8, 4 and 4 respectively.

Once the RTO limit is exceeded, a suggested action is to double the RTO:

$RTO = RTO * 2$

The message retransmission can be implemented using a thread that sleeps until the nearest retransmission time. The pseudo-code for sending messages is as follows:

```
SendMessage(Msg)
{
    SendUDPMessage(Msg)
    RTO = SRTT + (VARRTT * VARINC))
    Add Msg to retransmission list
    Activate retransmission thread event to recalculate the
    smallest RTO
}
```

The retransmission thread is presented below:

```
RetransmissionThread()
{
    Infinite Loop
    {
        If retransmission list is empty
            sleep time = infinite
        else
            sleep time = smallest RTO from list
        result = WaitEvent( thread activation, sleep time)
        if result == time limit expired
        {
            SendUDPMessage(Msg)
            RTO = RTO * 2
            Reposition the message in the list
        }
        else
            Deactivate the retransmission thread event
    }
}
```

WaitEvent is responsible for making the thread sleep until a thread activation event is flagged or a time limit is surpassed.

4. Technologies for I/O Processing of Multiple Sockets in Servers

Massively multiplayer servers need to process I/O for thousands of players and, in order to be effective, they need to use the most efficient technologies that are available.

The usual methods for processing I/O for multiple sockets are: (a) the use of blocking sockets with multiple threads, with a thread for each client; (b) the use of the blocking function `select()`. Both methods

have shortcomings and are not designed to handle the kind of workload that a massively multiplayer server must handle.

More efficient methods exist, but they come with the loss of portability. The most efficient methods are operating system dependent.

4.1 IO Completion Ports

This method is suggested as the most efficient method for processing multiple sockets in Windows [Jones et al., 1999].

A completion port is a list in which the operating system adds notifications of finished asynchronous overlapped operations. An application using completion ports generally creates a thread to handle these notifications. For multi-processor architectures, one thread is usually created for each processor to reduce context switching.

To create a completion port the following function must be used:

```
HANDLE CreateIoCompletionPort(HANDLE
   file_handle,
   HANDLE existing_comp_port,
   DWORD completion_key,
   DWORD num_of_conc_threads);
```

This function is used for two purposes: to bind a socket to a completion port, and to create the completion port. When this function is called to bind a socket to a completion port, all finished overlapped operations of the socket are notified in the completion port. When this function is used with the purpose of creating a completion port, the following actions should be considered: the first parameter must be defined as `INVALID_HANDLE_VALUE`; `existing_comp_port` must be defined as `NULL`; and `completion_key` must be ignored. The last parameter `num_of_conc_threads` defines the number of concurrent threads that can be executed in a completion port. Setting this parameter to zero will allow the same number of processors as threads to be executed in the completion port.

Below is an example of a completion port creation:

```
HANDLE comp_port = CreateIoCompletionPort
   (INVALID_HANDLE_VALUE, NULL, 0, 0);
```

To bind a socket to the completion port, the socket handle must be placed as the first parameter. The second parameter receives the completion port handle. The third parameter is a key that is sent to a thread that captures a notification for this socket. This parameter can be used to send a pointer related to the client data of the socket, for example:

```
CreateIoCompletionPort( (HANDLE) socket,
   comp_port,
   (DWORD) client_data_pointer,
   0);
```

In order to catch the notifications placed in the completion port the following function call must be used:

```
BOOL GetQueuedCompletionStatus
   (HANDLE comp_port,
   LPDWORD num_of_bytes_trans,
   LPDWORD completion_key,
   LPOVERLAPPED * overlapped,
   DWORD milliseconds);
```

The first parameter is the completion port that will supply the thread with notifications. `num_of_bytes_trans` returns with the number of bytes transferred in the I/O operation. The `completion_key parameter is the same as the one in CreateIoCompletionPort`. The `overlapped` parameter receives a pointer to an `OVERLAPPED` structure. This parameter is passed to the completion port when an overlapped operation is performed. The `milliseconds` parameter defines the time in milliseconds the function must block. Setting this parameter to INFINITE makes the function block eternally until an event happens. If the function returns a value that is different from zero, the operation will be considered successful.

The following pseudo-code shows how to make use of the API:

```
Function()
{
    Create the completion port
    Get the number of processors
    For each processor
        Create a notification handler thread
    Create an overlapped  socket (this can be done using the
    common socket() function)
    Bind the socket to the completion port
    Make an overlapped operation with the socket
}

NotificationHandlerThread()
{
    Infinite Loop
    {
        Catch a notification from the completion port
        If the operation was successful
            Process the operation
    }
}
```

Sometimes it is useful to receive extra information about a finished operation. For example, it could be helpful to know if the operation was either sending or receiving bytes. It is possible to do that by working with the pointer to the OVERLAPPED type sent to every overlapped operation. Below is an example:

```
struct Operation
{
   OVERLAPPED ov;
   int type;
};

Operation op;
```

```
op.type = SEND_OPERATION;
ZeroMemory(&op.ov, sizeof(OVERLAPPED));

WSASend(socket, buffer, 1, &bytes_sent,
   flags, &op, NULL);
```

When the send operation above finishes and `GetQueuedCompletionStatus` returns, it will be possible to verify the type of operation with the overlapped pointer as shown next:

```
NotificationHandlerThread()
{
   …
   Operation * op;
   GetQueuedCompletionStatus(comp_port,
      &bytes_trans,&key,
      (OVERLAPPED**)&op,
      INFINITE);
   If (op->type == SEND_OPERATION)
      Handle the send operation
}
```

5. Techniques for Distributed Virtual Environments

There are several techniques that can be applied to optimize the server processing. These techniques can greatly increase the number of players that can be connected simultaneously to a distributed virtual environment.

5.1 Goal Based Dead Reckoning

Dead Reckoning is a technique were the current position of an object is estimated based on previous information about that object [Singhal et al., 1999] [Singhal and Cheriton, 1994]. For example: based on the last position and direction of a ship, her current position is estimated using the formula below:

*current ship position = last ship position + last ship direction * time passed since last information*

An improvement to Dead Reckoning is the Goal-oriented Dead Reckoning proposed by Szwarcman et al. [2001]. In this type of Dead Reckoning, the clones in each client are autonomous objects that are free to act based on their goals. Messages are sent only when goals are deviating beyond certain limits. The goals can be anything: moving to a position, attacking an enemy, etc. Only the goals are transmitted through the network and this can result in a good reduction of traffic. Goal-oriented Dead Reckoning is not adequate for all types of games. In action games, for example, where the movement is generally controlled completely by the player, this type of Dead Reckoning is difficult to apply. Goal-oriented Dead Reckoning is good for massively multiplayer games that use a movement similar to Real Time Strategy Games, where the player only select the location he/she wants to be and then the selected unit moves by itself to the specified location.

5.2 Command Time Synchronization

Command Time Synchronization is a "movement prediction technique" that synchronizes objects of a distributed environment to be in the right place at the right time, even with the presence of latency [Alexander 2003].

Suppose a player wants to move to a certain location in the world. The player client sends a message to the server telling about the movement and then the server sends a message to all other clients about the player movement. There is some time elapsed from the moment the client sends the message to the moment the server receives the message. There is even more time elapsed in the moment the other clients receive the player movement. So, if the player starts moving right away after sending the movement message, he/she will arrive at the destination before the server and possibly much before the other clients. This leads to a great divergence in local game states between the peers of a distributed environment.

Command Time Synchronization is a "movement prediction technique" that synchronizes the players by using a shared global time and by modifying the player speeds based on the estimated moment of the arrival of the player in a particular destination. Below is an example:

In Figure 1, Client 1 requests to move a tank to the position 30. The tank is currently at the position 0 and it starts moving at speed 10. A message is sent to the server and it arrives at time 1 second. The server starts moving the tank to the position and it sends a message to all players (including Client 1) that the tank should arrive at position 30 at second 4, since the tank speed is 10 and it takes 3 seconds to arrive at the destination (Figure 2).

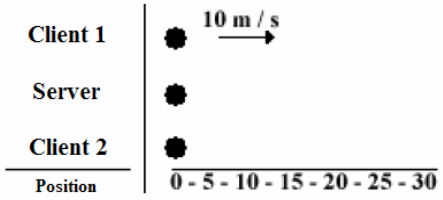

Figure 1: Second zero, tank at position 0.

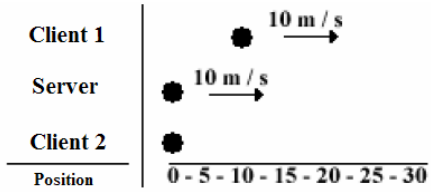

Figure 2: Second One, message arrives at server.

In the above example, the message takes one second to arrive at each client. All clients receive the message "tank from client 1 arrives at position 30 at second 4" and adjust the tank speed to meet the time

scheduled. In Client 1, the tank is already at position 20 since 2 seconds have passed. Then, the Client 1 changes the speed to 5 allowing the tank to reach position 30 at second 4. On the other clients, the tank speed is set at 15 allowing the tank to reach position 30 in two seconds (Figure 3).

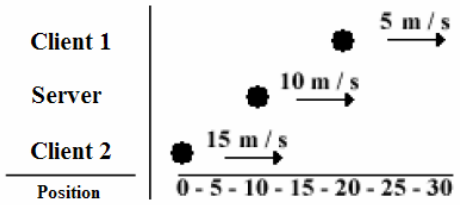

Figure 3: Second Two, message arrives at clients.

5.3 Communication Culling

Usually it is not necessary to send information to a client about another player that cannot be easily perceived, e.g.: a player in another corner of the virtual world. The client probably will not be able to see the other player, so there is no need to send information about its movement.

Computer graphics has often to handle a similar problem. It is generally very expensive to send all the data of a scene to the GPU for rendering; so, techniques for only sending what the camera can see were developed. A lot of hierarchical data structures from Computer Graphics can also be used for communication culling; a lot of them can be found in [Möller et al. 1999].

A simple data structure that allows fast detection of objects near a player is the grid. A grid divides the world in several rectangles of equal size. Each rectangle stores information about the objects that are occupying its space. The rectangle where an object is located can easily be calculated by the following expressions:

rectangle x = object pos x / rectangle width
rectangle y = object pos y / rectangle height

In order to obtain the objects near a player, it is a simple case of getting the objects in the rectangle occupied by the player and possibly the objects in the rectangles considered close. This region where the objects are considered close and possibly perceptible by the player is called “area of interest”. Figure 4 shows the grid culling technique (the black objects are being sent to the player, while the white objects are not).

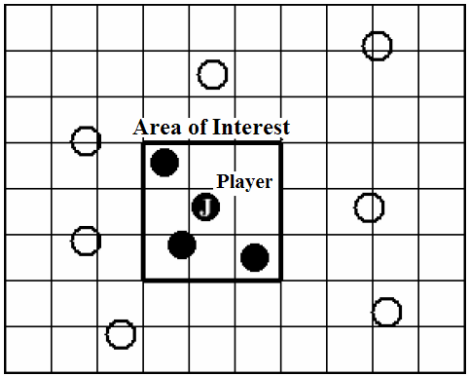

Figure 4: Grid Culling

5.4 Load Balancing

To process a massively multiplayer world is not a task for a slow server. In order to handle this kind of workload it is useful to break the world processing between multiple servers, allowing the use of single CPU computers in the processing.

When using multiple servers to handle the processing it is important to distribute intelligently the workload between the processors, otherwise a single processor can become overloaded and degradation of the player experience starts taking place.

One way to balance the workload is to break the world in rectangles leaving each server responsible for processing everything that happens inside a specific rectangle. While a player is inside a rectangle, he/she will only communicate with the server in charge. The moment the player passes to another rectangle, another server takes charge and the player starts communicating only with this new server.

It takes some time to connect to the new server, so it is useful to extend the rectangles frontier so that the client enters the new server rectangle already connected and receiving information about objects in the new area. To make this work properly, during frontier travel the client will be connected to the two servers of the frontier. This technique is shown in figure 5.

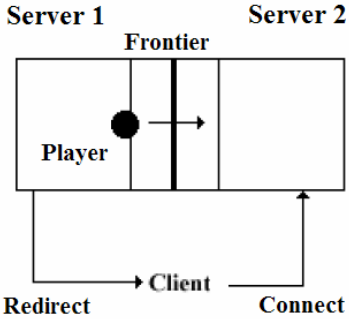

Figure 5: A client traveling through a frontier.

The problem with using rectangles is that in the corners the client will be connected with up to four servers, as shown in Figure 6.

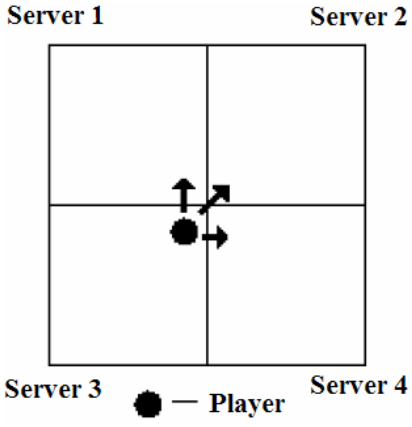

Figure 6: Client connecting to four servers in a rectangle divided world.

One alternative to the use of rectangles is using hexagons. When the world is divided in hexagons, the clients are never connected to more than three servers (Figure 7).

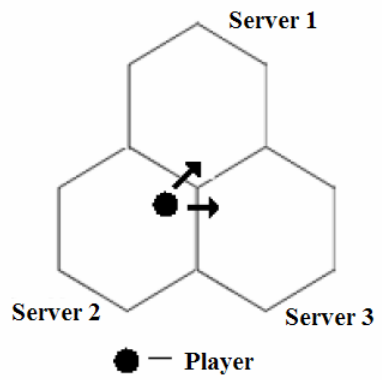

Figure 7: Client connecting to three servers in a hexagon divided world.

An improvement in the load balancing is to consider dynamic areas of influence, in order to avoid an excess of players in a single region. These areas can be implemented by moving frontiers.

Other ideas for load balancing can be found in [Kozovits, 2004].

5.5 Level of Detail

Level of Detail is another technique that is also used in Computer Graphics [Singhal et al. 1999]. The idea is that there is no need to send a lot of information to the player about objects that hardly affect the player's experience.

Usually the objects that have greater effect on the player's experience are the objects that are near him/her. So the update rate of those objects should be higher than the update rate of objects that are far away.

Structures like the grid used for communication culling explained in the session 5.3 can help in the choice of what objects should receive greater update rates. For example: objects inside the player rectangle receive greater update rate then the objects in the neighbor rectangles.

5.6 Message Aggregation

Another technique that allows the reduction of bytes transferred is message aggregation [Singhal et al. 1999]. If there is a group of messages that need to be sent to a client, it is better to join them and send a single message than to send a large number of small messages.

This is because messages sent using UDP have a header part of 28 bytes and messages sent using TCP have a header part of 40 bytes. So, the lesser the number of messages sent to a client, the smaller is the number of bytes wasted on header parts.

6. Conclusion

This work tries to group the most important techniques that massively multiplayer games must make use in order to use computer resources efficiently.

The literature on this subject is still scarce and more interdisciplinary research is required, especially in the area that overlaps distributed programming and computer network with computational geometry and computer graphics.

An important topic missed in this paper is multicasting. The use of multicasting would allow a great reduction in the number of messages. Information on this topic can be found in [Day, 2004] [Lukianov, 2001].

Techniques on security and cheating are not considered in this paper, because the focus here is on communication. Also, no comments are made on simplified techniques that although less robust can lead to good results in a short development time. In this particular the work by Feijo and Binder [2004] is recommended.

Acknowledgements

The authors would like to thank all the colleagues at VLab/IGames for their help and dedication in making a great work and research on games. Also the authors would like to thank Richard Garriot (a.k.a. "Lord British") for his inventions that started the massively multiplayer frenzy.

References

ALEXANDER, T., 2003. Massively Multiplayer Game Development. Massachusetts: Charles River Media.

DAY, J., 2004. *Introduction to Multicasting*. Freshmeat Tutorials. Available from: http://freshmeat.net/articles/view/1185 [Accessed 30 Aug 2006].

FEIJO, B., BINDER, F.V., 2004. Conceituando e resolvendo pragmaticamente os problemas mais críticos de um MMORPG. *Scientia*, 15 (2), 158-165.

GDC, 2006. *Engineering Issues in Multiplayer Game Development*. Game Developers Conference. Cited in: www.gdconf.com [tutorial not published] [Accessed 30 August 2007].

IEEE, 1993. Protocols for distributed simulation applications: entity information and interaction. IEEE Standard 1278.

JONES, A. OHLUND, J., 1999. Network Programming for Windows. Washington: Microsoft Press.

LUKIANOV, D., 2001. *Advanced WinSock Multiplayer Game Programming: Multicasting*. GameDev.net. Available from: http://www.gamedev.net/reference/articles/article1587.asp [Accessed 30 Aug 2006].

MACHADO, L., 2005. Técnicas de Ambientes Virtuais Distribuídos para Jogadores em Massa. MSc Dissertation, Dept. of Informatics, PUC-Rio, Rio de Janeiro, Brazil. [in Portuguese].

MICROSOFT, 2005. *Socket overlapped I/O versus blocking/nonblocking mode* [online] Microsoft Help and Support, Article ID 181611, Rev. 3.1. Available from: http://support.microsoft.com/default.aspx?scid=kb;EN-US;q181611 [Accessed 30 Aug 2006].

MOLLER. T., HAINES, E., 1999. Real Time Rendering. Massachusetts: A K Peters.

ORAM, A., 2001. Peer-to-Peer: Harnessing the Power of Disruptive Technologies. O'Reilly Media.

PARTRIDGE, C., PINK, S., 1993. A faster UDP. IEEE/ACM Transactions on Networking, 1(4): 429-440, 1993.

PENTON, R., 2003. MUD Game Programming. Game Development Series (Ed. André LaMothe), The Premier Press.

SINGHAL, K.S., CHERITON, D., 1994. Using a position history-based protocol for distributed object visualization. Technical Report, Dept. of Computer Science, Stanford University.

SINGHAL, S., ZYDA, M., 1999. Networked Virtual Environments: Design and Implementation. Boston: Addison Wesley.

SNADER, J. C., 2000. Effective TCP/IP Programming: 44 Tips to Improve your Network Programs. Boston: Addison Wesley.

STEVENS, W. R., 1999. TCP/IP Illustrated Volume 1 : The Protocols. Boston: Addison Wesley.

SZWARCMAN, D., FEIJO, B., COSTA, M., 2001. Goal-oriented dead reckoning for autonomous characters. *Computers & Graphics*, 5 (6), 999-1011.

KOZOVITS, E.L., 2004. Otimização de mensagens e balanceamento de jogos multi-jogador. PhD Thesis, Dept. of Informatics, PUC-Rio, Rio de Janeiro, Brazil. [in Portuguese].

KUHL, F., WEATHERLY, R., DAHMANN, J., 1999. Creating Computer Simulation Systems: an Introduction to the High Level Architecture. Prentice Hall, New Jersey.