A Multiplayer Real-Time Game Protocol Architecture for Reducing Network Latency

Yong woon Ahn, Albert Mo Kim Cheng, *Senior Member*, IEEE, Jinsuk Baek, *Member*, IEEE, and Paul S. Fisher

Abstract — *Multiplayer, real-time games have become a huge commercial service. Because of this success, the need exists for a more efficient and reliable game server solution. Such games are a kind of soft real-time systems because a game server has to respond to requests from many clients, compute the action of a non-player character (NPC), and filter the data from each client within specified time constraints. Some client and server tasks are periodic and some are not. To meet these timing constraints, we propose a task scheduling policy. Our solution includes two approaches: (1) the STU segment queue approach to convert sporadic events from game clients into serialized periodic event groups; and (2) the spatial domain approach to reduce queuing delay caused by the network outgoing event queue. To implement our solution on the JAVA framework with the non-real-time JAVA virtual machine, we propose a new client/server architecture for a scalable game server that in addition, will reduce the frequency of garbage collection by reusing server resources*[1].

Index Terms — Real-time game server, Segment queue, Spatial domain, Garbage collection.

I. Introduction

Multiplayer games with their associated online, real-time requirements for information must have assurance that the game actions are delivered within the real-time constraints established by the play. Delay in this data can make the game no longer playable. Thus, multiplayer, real-time games have strict demands on the underlying network and require a low latency connection in order to meet the quality of service (QoS) constraints. There can be some delay tolerance in this environment, but overall system performance is still mandatory [1].

The architecture of such games is similar to other client/server architectures using UDP networks such as required by video/audio streaming architectures. UDP with timestamps [2] can be a useful way to implement a reliable server. Such an example is available in [3]. These stand-alone simulators can be connected to each other and controlled through unpredictable user events operating under a centralized server application. Other applications of such solutions could be for controlling multiple, physical devices having sensors operating over a networked environment such as a networked robot [4]. Such games also require tasks:

1. arriving from the player to be organized and executed by priority to best maintain a QoS;
2. to be executed with a specific scheduling policy;
3. that are periodic or sporadic to update the game world, filter client data, and respond to timed actions;
4. to be abandoned when there is insufficient execution time available.

The game server described has been implemented using JAVA [5] because of its hardware and operating system independence. Other advantages from Java include server expansion and migration to other systems, since it is easy to integrate many server side technologies such as web services, web applications, enterprise applications, and management and security protocols. In the server side application, servers based upon the JAVA platform are at least as fast as other C or C++ based servers since the release of the JAVA Development Kit (JDK) 1.4. We use the New Input/Output (NIO) package including the *non-blocking I/O* technology to achieve this required speed. With NIO [6], JAVA applications can access the main memory directly providing a fast execution. Also, if we want to use the library implemented by C or C++ such as 'lib' or 'dll', the JAVA Native Interface (JNI) provides a solution.

II. Consideration for Real-time Multiplayer Game

A game client application is multifaceted including: 1) behaviors predefined by developer; 2) a virtual game world that players interact with; and 3) events caused by players' actions. Such actions cause the application to gather events from all clients, recalculate the game circumstance, and then send the updated information of the virtual world to all participants. Developers also make decisions about: 1) the maximal number of connected clients per server; and 2) how many servers can be managed; and the minimum network bandwidth required to play the game.

[1] This work was supported in part by the National Science Foundation under Award No. 0720856.

Yong woon Ahn is with the Department of Computer Science, University of Houston, Houston, TX 77204 USA (e-mail:yahn@uh.edu).

Albert Mo Kim Cheng is with the Department of Computer Science, University of Houston, Houston, TX 77204 USA (e-mail:cheng@cs.uh.edu).

Jinsuk Baek is with the Department of Computer Science, Winston-Salem State University, Winston-Salem, NC 27110 USA (e-mail:baekj@wssu.edu).

Paul S. Fisher is with the Department of Computer Science, Winston-Salem State University, Winston-Salem, NC 27110 USA (e-mail:fisherp@wssu.edu).

Contributed Paper
Manuscript received October 15, 2009

Since the player input is random and there are other non-player characters (NPC) that should be managed, the game server has a plethora of tasks to manage and schedule. Among them, the first step is to classify queued tasks by their properties. User actions are usually handled as sporadic tasks with deadlines due to their randomness. Conversely, events updating the game world can be periodic tasks. For example, NPC actions might be sporadic tasks or periodic tasks with deadlines because spawning a NPC is periodic but NPC actions are sporadic.

A. Service Time Unit

We define a *Service Time Unit* (STU) as the shortest time interval needed to update the game world and process the accumulated requests from clients and all NPC actions. The server broadcasts updated events to all clients, or multicasts to selected clients, waiting for responds, not affecting other players. In addition, for a fluid game experience, a game server has to create an illusion that there is no time lag while playing. For example, a First Person Shooting (FPS) game [7],[8], the server updates the game world every 30 to 40 ms. Figure 1 shows a simple schedule for one STU containing various kinds of tasks.

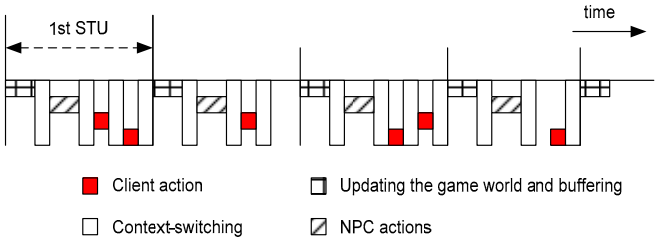


Fig. 1. Example of STU

Depending on the current game situation, the number of tasks and computation time during one STU will be altered. If most of the players are scattered in the game world and only few avatars (the virtual representations of participating users) are visible to each other, then few interactions have to be processed within one STU. Otherwise, more operations have to be calculated if each client and server initiates an event at the same time. This can be expressed by the inequality: $[C_m m+C_n n+C_l l+T_c(m+n+l)] \le t \bullet STU$ where m is the update number for the game world, n the number of actions from clients, and l the number of NPC actions, C_k is computation time for k, T_c is context switching time, t is t_{th} STU and every event has the same deadline as $t \bullet STU$.

We also assume that all server side actions have higher priority than a client's event. A game server's action is critical because missing the deadline at the server side application may affect all game participants; therefore, we need to guarantee that all clients receive server actions within the deadline. Otherwise, if a client's action or request misses the one STU time, the server would abandon its processing because we cannot guarantee the order of the packets if UDP is used.

B. Recovering from Congested Game Processing

When a task misses its deadline, we must abandon this task to avoid a situation of *congested game processing*. Therefore, we need to define the policy to choose tasks to be abandoned. We define the *ignore point* as the time after which a game server does not execute the task having a deadline, $t \bullet STU$, as: $t \bullet STU - (C_n + T_c)$. Because there may be packet fragments in the network buffer, we must consider the packet fragmentation of the client's action to determine the actual ignore point.

C. Reducing Garbage Collection Frequency

One of the consequences of a very busy dynamic system is that it creates many new objects with short lifetime. A heavily loaded system (20K ~ 100K message/sec) can spend as much as 30 seconds with all the application threads suspended while *full garbage collection* or *minor garbage collection* executes. This is detrimental to a high performance system [8].

Garbage collection is impossible to predict, and it will occur because there are many instances that are not directly initiated by a developer who uses external components such as a database connection, an XML parser, etc. In order to reduce the likelihood of garbage collection, we should not create a brand new thread which is not predefined. If we can use a finite number of threads, the server could reuse a thread already created. Our game server uses the *thread pool* to handle all game clients that are connected to the game server. We will show the results of this implementation strategy when we consider the simulation of the game server.

III. Client/Server Architecture for Real-time Multiplayer Game

Fig. 2 shows the architecture proposed for the environment supporting a large number of players with the task segment that can use conventional scheduling policies.

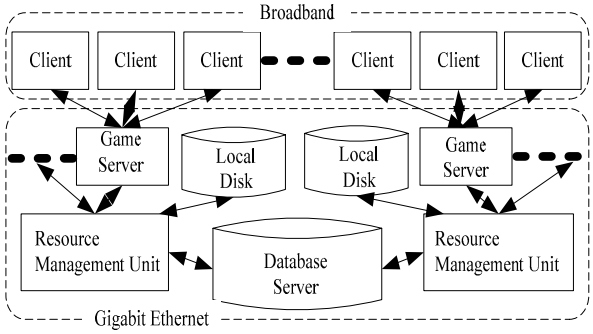


Fig. 2. Client/Server architecture for multiplayer, real-time game

There are multiple game servers managing the partial game world or game rounds for clients and accepting connection requests from game clients and interacting with the resource management server (RMS). Each RMS determines whether requests from the game server are stored

into the database server immediately or not. For this operation, the RMS has a local storage device to store requests which can be stored after finishing the game round. The database server can store game player's information and the game world status such as the item inventory, game world objects, game records, and so on. We assume that game clients are using a broadband Internet connection (at least ISDN). The server clusters are grouped within the gigabit Ethernet environment as shown in Fig. 2.

Fig. 3 illustrates the modules in the game server. The events from clients are accumulated into the *incoming event queue* before determining any abandoned task. To check the deadline and abandon the task, the deadline checking module compares the deadline and current STU clock, and if appropriate assigns the task to the correct STU segment which is not in the critical section. Each STU segment cannot be interrupted by another STU segment while currently running in the critical section.

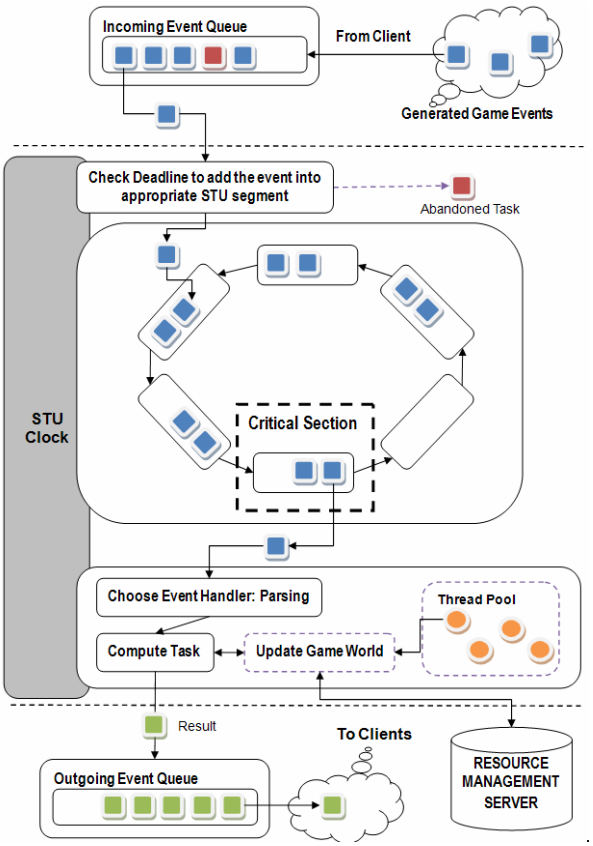


Fig. 3. Game server modules

A. Incoming Event Queue

The *incoming event queue* storing the events from clients is similar to a network buffer with a *priority inversion* [10] scheme. Although the basic schedule policy is FIFO, the *incoming event queue monitor* can reorder the events when a higher priority event arrives. Usually the shortest deadline event and a server's event may have the highest priority, but starvation can be a real problem for events having lower priorities. To solve this problem, we use a solution from the *Real-Time Specification for JAVA* (RTSJ) [11]. The maximum event priority is assigned to the front of the incoming event queue not to be interrupted by another arriving higher priority event.

We assign priorities for the game events because of their potential for global game impact followed by their contribution to updates of the current game world. For example, if one game player modifies a part of the game world, this event should have higher priority than other game events such as 'chat' or 'movement'. This approach has the problem that a high priority event may not be executed immediately. The incoming event queue monitor module manages this inversion process, and tries to avoid the starvation situation. The best solution would be to expand our single game server approach to a multiple server approach for the single incoming event queue.

B. STU Segment Queue Approach

Since this system is an unpredictable event driven system, it is also very hard to schedule events using conventional scheduling techniques such as the earliest deadline first (EDF) or the least laxity first (LLF). Therefore we convert those sporadic events into the periodic events to apply conventional scheduling policies. To achieve this objective in the server, we propose a new approach: the STU segment queue approach. If the event would not violate the deadline, the action for this event could be assigned with a deadline to the appropriate STU segment which is an instance of the server STU clock. Also the appropriate STU segment must have available time periods for storing it. For example, if there are no available time periods to execute the event handlers within the certain STU segment, the event may be abandoned, even if it has a valid deadline under the current STU clock. We need to mention that the STU segment, which is currently inside of the critical section, cannot be considered for accepting a new event from the incoming event queue. The most important advantage of the STU segment queue approach is that segments located in the queue can be scheduled using conventional scheduling policies for periodic tasks because we can assume each STU segment as one task for them, and then we can calculate the worst case computation time for each segment. This approach also allows us to monitor current server status and predict the next status.

C. Event Handler

Each STU segment can be processed by a thread containing event handlers for each event. First, the event handler management module parses the event packet and calls a pertinent event handler. After this the thread pool management module picks an available thread from the thread pool. If there is no available thread, the STU segment must wait for the previous job to finish under control of the currently running thread. The picked thread can be the event handler to update the game world, interact with other clients, and update the database game server with the final result queued to the outgoing event queue. To assign the execution

priority for each event handler, the game server must have a constant computation time for each event handler to use conventional scheduling policies.

D. Spatial Domain Approach

To respond on time with the updated result by the server, the server should decrease the number of result packets to be sent since a queuing delay may engender missing a deadline. The game server is responsible to notify updates of the game world to all clients participating in the game. However, we consider a network environment as a set of strictly limited resources such as a network bandwidth, throughput, latency, and so on. In a view of the real-time requirement, these limited resource problems might impact the execution of events within a specific deadline from the server. To overcome the limited network resource problem, we propose the spatial domain approach to select destination game clients. The idea is this, when one player wants to update status, the client application must determine whether the event directly affects other players or the game world. If one player's action directly affects another player, we can assume that this event is applied to players who are located in the spatial domain of a source game client in the game world. The conventional solution is to notify all players, but this wastes resources, since a player may not be participating in the local view of the change. The spatial domain must be larger than a screen resolution which can be shown to game players as we can see in Fig. 4.

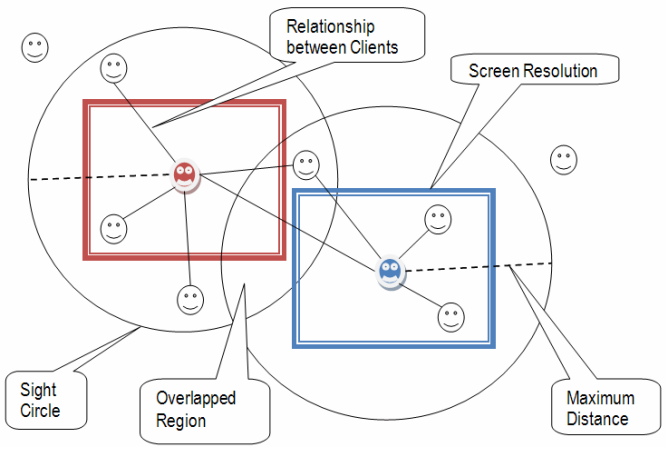


Fig. 4. Spatial domain for selective multicast

The maximum distance of the sight domain is a constant value which can be determined as a radius of the sight circle. To avoid additional calculations by the game server, each client maintains the location data of all other players within the sight circle to calculate the distances between them. When sending the event to the game server, the game client application also attaches the game client's identifications with the events. After receiving the events from the clients, the server simply multicasts the events to selective clients registered at the server. A primary benefit of this approach is that we can efficiently use network bandwidth for the game server because the server does not need to broadcast the updated events to all clients. In other words, the server would have more concurrent user capacity because our approach also uses the non-blocking I/O enabling all clients to register on the channel using the outgoing event queue. Managing the outgoing event queue to avoid missing a deadline in the server is critical.

E. Outgoing Event Queue

Every event must be stored into the outgoing event queue before responding to an event from the game server to the game clients. The server has the STU clock to send or receive the events periodically, and to monitor the outgoing event queue, we use the monitoring module. This module will measure the current use of the outgoing event queue and force a developer to determine the optimal buffer size for their real-time game. If the outgoing event queue overflows from the game data being sent to the game clients, the developers must alter the queue size or revise the packet to reduce its length.

F. Client Module

We also note that the client application should have the same architecture as the server application minus the STU segment queue. The processing is almost identical except the deadline module filters the events that miss the deadline prior to assigning a thread from the pool based upon the deadline data contained in the packet. After assigning the thread, the event handler updates the game world to display the various movements from all relevant players or the game server.

IV. Experimental Results

In order to measure the efficacy of our design, we use a non-blocking I/O mode and the thread pool, and then we measured the garbage collection frequency, and tested with various numbers of predefined threads in the thread pool; and lastly, we tested the outgoing event queue using the broadcasting and our multicasting method. The simulation was done on a workstation with the 2.0 GHz 64bit dual core processor; 2Gbyte RAM; minimum heap size of JVM being 2Mbytes and maximum heap size being 5Mbytes.

- Purpose: *Comparing a Non-Blocking I/O Model using NIO with One Thread per One Connection Mode*
  1. Objective: minor and full garbage collection measurement.
  2. Use between 10 and 50 clients, each sends the same message (12 bytes) to the server 30 times every 0.3 sec.
  3. Each client repeats connecting to the server and sending the message 10 times.
  4. For the one thread per one connection model: server creates 500 threads for 50 clients; non-blocking server creates one thread to receive and send the message.

- *Result*

  From Fig. 5, using a NIO server model is more useful in reducing the frequency of garbage collection (GC) than a thread per connection (TPC) model.

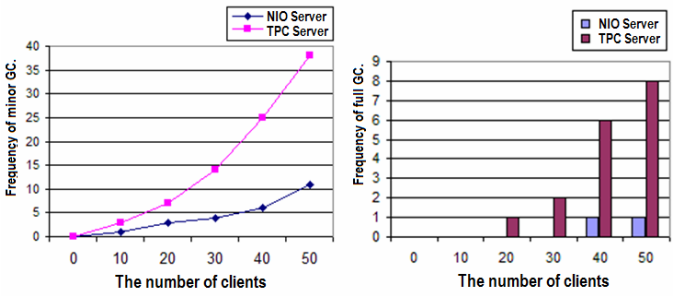


Fig. 5. Results comparing a TPC model to a NIO model using non-blocking I/O

- Purpose: *Impact of Using Thread Pool to Reuse the Resources on Minor Garbage Collections*
  1. Assumption: It is impossible to predict, due to the randomness of the game environment, the number of players to connect, play the game, finish their session simultaneously, and their work load. Again we use a multi-threaded server with non-blocking I/O.
  2. Objective: Using a thread pool, measure frequency of minor garbage collection.
  3. Establish 100 clients sending 100 messages (12 bytes of each), all receiving the same message from the server.
  4. Upon receipt of message from the game clients, server creates and deletes the 10,000 dummy messages (12 bytes of each) before sending the message the server received. Dummy messages are the garbage to be collected.

- *Result*

  We see an increase in the number of the minor garbage collections when increasing the number of threads in the thread pool. This is shown in Fig. 6. In Fig. 7, we show the response time for clients to receive the same message (12 bytes) from the server and to finish their connections. We expected the response time would decrease while increasing the number of threads in the thread pool. We could only get partial results after three threads since the processor could handle all of the messages with only two threads in the pool.

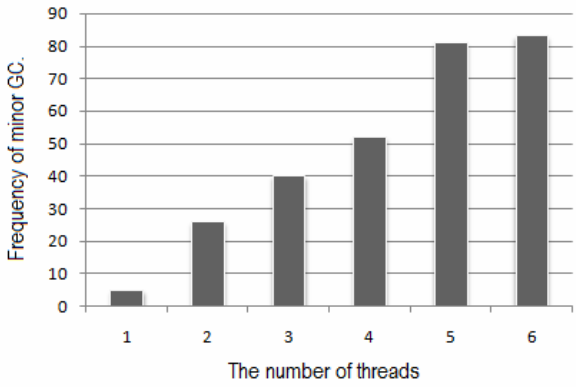


Fig. 6. The number of the minor garbage collection by the various numbers of threads

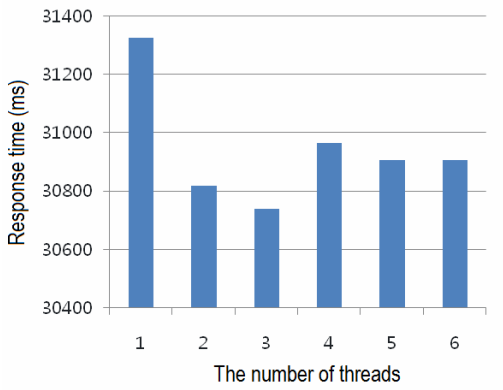


Fig. 7. The response time from the simulation

- Purpose: *Experiment to Measure Use of Outgoing Event Queue*

With the conventional broadcasting approach, the game server sends all updated game world information to all game participants. With the multicasting approach for selected clients, the game server sends the updated information to selected clients which are located within the spatial domain, and selected by interaction with the event protocols.

1. Assumptions:
   1) The game client must generate similar game events to the actual multiplayer real-time game like Quake 3 [8] where the most frequent event is a 'movement'.
   2) We must implement the game client with more demanding conditions since we were able to only experiment using the multicasting approach with a small number of clients. The events, generated more frequently than actual user play, are randomly generated with 300 ms time interval by NPCs.
   3) We use only fixed length packets for each event to simplify monitoring of the outgoing event queue. We do not generate a 'chat' event to interact with other players because it might not be a fixed length packet.
   4) Game client and server communicate with each other using UDP with the deadline which can be assigned by the game server using the STU clock.
   5) To monitor the outgoing event queue, we use the fixed size of a byte buffer whose capacity is 2048 Mbytes.
2. Objective: Compare results with two versions for a message transmission approach: the conventional broadcasting approach and our proposed multicast approach using the spatial domain.
3. We installed the client application with an NPC on five different Java Virtual Machines using default options to execute JAVA applications with 5 Mbytes of maximum heap size. Each game client sends 'connection', 'disconnection', 'movement' 'attack', events to the game server with a variable but mean 300 ms interval. Each client is spawned at the same location when the game is started.

4. One game server maintains the STU clock with a 30 ms interval, and there are ten available threads in the thread pool. The STU clock thread has the highest priority to periodically generate the clock signal.

- *Result 1*:
  In the case of the conventional broadcasting approach, since the server must broadcast all updated information of the game world to all connected game clients, the outgoing network buffer experiences a larger amount of use than the outgoing event queue as shown in Fig. 8. We have an average use of the outgoing event queue of approximately 440 bytes per 30 ms which is the interval of one STU.

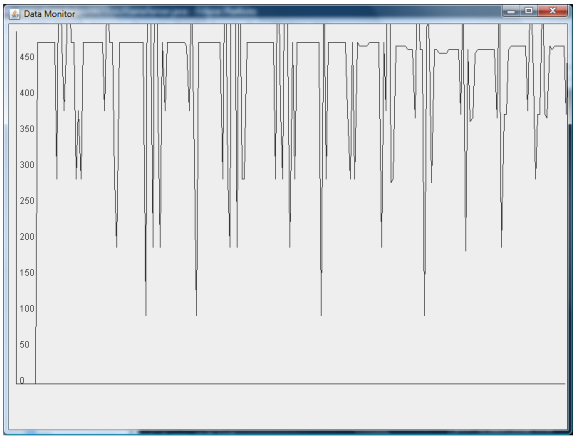


Fig. 8. Using conventional broadcasting method: average use of outgoing event queue is approximately 440 bytes/STU

- *Result 2*:
  From the experiment using our multicasting approach the game server uses the outgoing event queue with a smaller amount of data being sent. In this case it is approximately 380 bytes per 30 ms. This is shown in Fig. 9. We can save about 24% of the usage of an outgoing event queue. In other words, we can build a more scalable server system, if the server uses the multicasting method using our spatial domain approach.

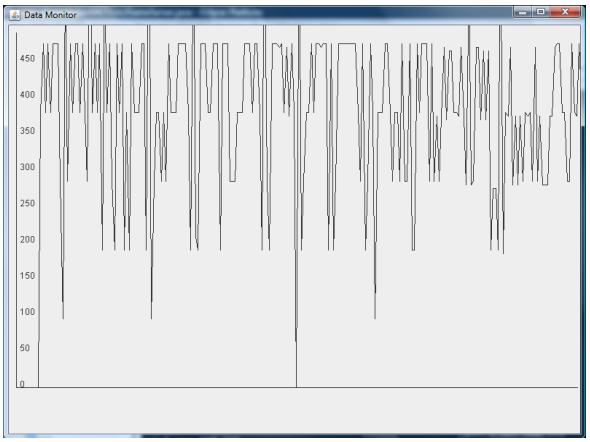


Fig. 9. Using new spatial domain approach: average use of outgoing event queue is approximately 380 bytes/STU

V. CONCLUSIONS

We proposed an architecture for a multiplayer real-time game server based upon the JAVA framework, where a multiplayer real-time game includes various task events that are periodic and sporadic. To perform scheduling, we needed to assign various priorities to the tasks. Basically, tasks from clients and sporadic tasks have lower priorities to avoid the case of *congested game processing*. We proposed a non-blocking I/O solution using a *Thread Pool* which we have shown to be better than the conventional client/server model in reducing the frequency of garbage collection. To show the change in garbage collection, we implemented two server programs with the conventional one thread per one connection and the NIO scheme.

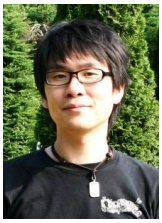
The *STU segment approach* allows the sporadic events from game clients to be converted into a group of periodic events that can be scheduled using conventional scheduling algorithms such as EDF or LLF, and this allows grouping of events with similar deadlines. Each STU segment has a fixed capacity to store event handlers. Since computation time for each event handler can be determined, the STU segment queue management module knows whether a pertinent STU segment is already full or not. As a future work, we are considering a new approach which enables the multiple STU segment queues to add a new event into appropriate queue among the STU segment queues having different priority levels.

We proposed a s*patial domain approach* to reduce the network latency which might be a serious obstacle in sending a real-time event within a specified deadline. Our spatial domain approach avoids the broadcast method: the game severs sends all updated messages to the game participant community. The spatial domain approach enables that the game server to select a group of destination game clients. From the experiment, our new approach reduces by a factor of about 24%, the usage of the outgoing event queue where game events are generated by a NPC behaving like an actual game player.

REFERENCES

[1] J. Muller, and S Gorlatach, "GSM: A Game Scalability Model for Multiplayer Real-time Games," *Proceedings of IEEE INFOCOM 2005*, pp. 2044–2055, March 2005.
[2] H. Schulzrinne, S. Casner, R. Frederick, V. Jacobson, RTP: A Transport Protocol for Real-time Applications. Internet Engineering Task Force, RFC 1889 (1996).
[3] B. Sweetman, "Ultimate Fighter: Lockhead Martin F-35 Joint Strike Fighter," Zenith Press, 2004.
[4] S. Wang, G. P. Liu, D. Rees, and M. Tan, "Development of Networked Robot Fish Control Systems," *Proceedings of the 6th World Congress on Intelligent Control and Automation* (WCICA), pp. 9317–9321, June 2006.
[5] C. S. Horstmann, and G. Cornell, "Core JAVA", Sun Microsystems Press, 2007.
[6] R. Hitchens, "How to Build a Scalable Multiplexed Server with NIO," *JavaOne Conference*, session TS-1315, May 2006.
[7] A. Abdelkhalek, A. Bilas, and A. Moshovos, "Behavior and performance of interactive multi-player game servers," *Proceedings of the 2001 IEEE International Symposium on Performance Analysis of Systems and Software* (ISPASS), pp. 137–146, November 2001.
[8] S. Honeywell, "Quake 3 Arena," Prima Games, 1999.
[9] D. F. Bacon, P. Cheng, D, Grove, M. Hind, V.T. Rajan, E. Yahav, M, Hauswirth, C. M. Kirsch, D. Spoonhower, and M. T. Vechev, "High-

level Real-time Programming in Java," *Proceedings of the 2005 International Conference on Embedded Software* (EMSOFT), pp. 10–22, September 2005.
[10] J. Wellings, and A. Burns, "Asynchronous event handling and real-time threads in the real-time specification for Java," *Proceedings of the 8th Real-Time and Embedded Technology and Applications Symposium* (RTETAS), pp. 81–89, September 2002.
[11] P. Mikhalenko, "Real-Time JAVA," O'Reilly, 2006.



Yong woon Ahn is a Ph.D. student of Computer Science at the University of Houston (UH). He is a member of the Real-Time Systems Laboratory at UH, and works for VNet as an AJAX developer. He received his B.S. and M.S. degrees in Computer Science and Engineering from Hankuk University of Foreign Studies (HUFS), Korea, in 2001 and 2003. He has developed a client/server system for the on-line multi-player Go game at Seloco Inc., Seoul, Korea, in 2003 and 2004. He has recently developed a geographical information system accepting the generic location data for Baker Hughes Inc., Houston, TX, in 2008. His research interests include mobile and wireless sensor networking, real-time systems, fault-tolerant computing, ubiquitous computing with embedded devices, and middleware for scalable network environment.

Albert Mo Kim Cheng received the B.A. with Highest Honors in Computer Science, graduating Phi Beta Kappa, the M.S. in Computer Science with a minor in Electrical Engineering, and the Ph.D. in Computer Science, all from The University of Texas at Austin, where he held a GTE Foundation Doctoral Fellowship. Dr. Cheng is currently a tenured Associate Professor in the Department of Computer Science at the University of Houston, where he is the founding Director of the Real-Time Systems Laboratory. He has served as a technical consultant for several organizations, including IBM, and was also a visiting faculty in the Departments of Computer Science at Rice University (2000) and at the City University of Hong Kong (1995). Dr. Cheng is the author/co-author of over 110 refereed publications in real-time/embedded systems and related areas, and has received numerous awards, including the NSF Research Initiation Award. He has been invited to present seminars, tutorials, and panel positions at over 30 conferences, has given invited seminars/keynotes at over 30 universities and organizations. He is and has been on the technical program committees of over 100 conferences, symposia, workshops, and editorial boards (including the IEEE Transactions on Software Engineering, 1998-2003). He is a Senior Member of the IEEE. Dr. Cheng is the author of the new senior/graduate-level textbook entitled Real-Time Systems: Scheduling, Analysis, and Verification (John Wiley & Sons 2002 and 2005).

Jinsuk Baek is Assistant Professor of Computer Science at the Winston-Salem State University (WSSU), Winston-Salem, NC. He is the director of Network Protocols Group at the WSSU. He received his B.S. and M.S. degrees in Computer Science and Engineering from Hankuk University of Foreign Studies (HUFS), Korea, in 1996 and 1998, respectively and his Ph.D. in Computer Science from the University of Houston (UH) in 2004. Dr. Baek was a post doctorate research associate of the Distributed Multimedia Research Group at the UH. He acted as a consulting expert on behalf of Apple Computer, Inc in connection with Rong and Gabello Law Firm which serves as legal counsel to Apple computer. His research interests include scalable reliable multicast protocols, mobile computing, network security protocols, proxy caching systems, and formal verification of communication protocols. He is a member of the IEEE.

Paul S. Fisher is R. J. Reynolds Distinguished Professor of Computer Science at the Winston-Salem State University (WSSU), Winston-Salem, NC. He is the director of High Performance Computing Group at the WSSU. He received his B.A. and M.A. degrees in Mathematics from University of Utah and his Ph.D. in Computer Science from Arizona State University. He has written and managed more than 100 proposal efforts for corporations and DoD involving teams of 1 to 15 people. He worked as consultant to the U.S Army, U.S Navy, U.S Air Force and several companies over the years. In the 1990's he commercialized an SBIR funded effort and built Lightning Strike, a wavelet compression codec, then sold the company to return to academe. His current research interests include wired/wireless communication protocols, image processing and pattern recognition.