Communicating Process Architectures 2006
Peter Welch, Jon Kerridge, and Fred Barnes (Eds.)
IOS Press, 2006

A JCSP.net Implementation of a Massively Multiplayer Online Game

Shyam KUMAR [a] [1] and G. S. STILES [b]
[a] *28 Gaslight Drive, Apt #7, South Weymouth, MA 02190*
[b] *Utah State University, Logan UT 84322-4120*

Abstract. We have developed a simple massively multiplayer online game system as a test bed for evaluating the usefulness and performance of JCSP.net. The system consists of a primary login server, secondary servers managing play on geographically distinct playing fields, and an arbitrary number of players. The basic structure of the game system is fairly simple, and has been verified to be free from deadlock and livelock using CSP and FDR. The JCSP.net implementation is straight-forward and includes over-writing buffers so that disconnected players will not block the servers and other players. Performance tests on local area networks under Windows demonstrate that five secondary servers easily support 1,000 machine-generated players making moves every two to five seconds. The player move end-to-end time was about 65 milliseconds, which is considered fast enough to support fast-action online games. Conversion from Windows to Linux required minimal effort; limited tests confirmed that versions using Linux alone, and Windows and Linux together, were also successful.

Keywords. Massively multiplayer online game, Java, JCSP.net, concurrency.

Introduction

The purpose of this project is to evaluate the usefulness of CSP-based tools in the design of a large distributed online game system and the performance of the JCSP.net package [1, 9, 10] in its implementation under Windows and Linux. Massively multiplayer online games (MMOGs) are an increasingly popular form of distributed online entertainment [2]. MMOGs are projected to be a $2.5 billion industry and to have nearly 114 million subscribers in 2006 [3, 4, 5]. Research papers on MMOGs are now appearing in technical journals and conferences [e.g. 2, 4, 6], and universities are adding courses on various aspects of gaming. In addition to recreation, online computer gaming also has ties to more serious applications: military organizations around the world train their personnel on computer-based war games [16, 17] to develop decision-making and leadership skills.

Distributed gaming systems involve a great deal of concurrency. Such systems are difficult to implement correctly without proper tools, and can have subtle (and serious) errors that may not appear until well after a product has shipped. Thousands of players around the world may be logged in to a game system at any one time, and a high degree of fault tolerance should be maintained: failures of individual components should not bring the entire system down.

We develop a reliable gaming system by first modeling the basic game structure in CSP [7]. We use the FDR model checking tool [8] to verify that the design is free of

[1] Corresponding Author: kumar.shyam@gmail.com

deadlock and livelock. The design is then implemented in Java using the JCSP.net library [1, 9, 10], which includes CSP-based features that support network communication among concurrent Java processes. Performance tests are then run, under both Windows and Linux, with up to 1,000 players and five secondary servers.

1. Background

1.1 Overall Design Considerations

Several issues must be addressed in the design of distributed massively multiplayer network games. We need to consider the large number of players our game might host, and must design the components accordingly. The system should be easy to expand as the number of players increases. The unpredictability of the internet, security, testing, and the need for continuous dynamic updates should also be considered.

We look first at possible server architectures. Most MMOGs are based on a client-server structure in which a number of players act as clients connected to a server that responds to the clients' requests. Some games also support peer-to-peer connections where communication between players is direct, bypassing the server; these systems tend be limited to smaller networks and may be less secure. The client-server model overcomes these limitations, but at higher cost and a more complicated network.

Compared to peer-to-peer networks, scalability is better in the client-server systems [11]. Many clients can connect to a server and communicate between each other via that server. The number of clients playing the game at any instance is limited only by the server capacity.

1.2 Server Issues

Traditional online games have client-server architectures with messages being passed through the server each time a player makes a move. Traffic at the server side is proportional to the number of players and can create congestion; distributed server architectures help to overcome these problems. Everquest uses about 40 independent servers, each able to handle nearly 2,000 players [14].

Client-server game architectures can be classified into two types, viz. zone-based and seamless [11]. In the zone-based approach the servers can be distributed around the physical world, and also distributed with respect to different domains within the game world; players may need to be aware of this structure. Seamless systems, on the other hand, hide all server details from the players.

Zone-based servers use a static server infrastructure that imposes arbitrary restrictions on players by coercing them into one shard (a superset of a zone) or another. These shards are further broken down into zones of a fixed size. Passage between zones is possible, but only by interrupting game-play while the new zone is loaded on the client. Each zone allows a maximum number of active players. If a particular zone server reaches its maximum occupancy or goes down, migration of clients can take place automatically. Migration of a player's data may also take place if the player changes his location within the game world; the game system will automatically move all player data to the new zone [12]. If zones are based on the location of the player in the physical world, the software can provide a list of nearby servers. This, in most cases, effectively balances the load on the system.

Seamless servers act as completely hidden systems wherein the server details are invisible to the players. Scalability and reliability is claimed to be better than in zone

servers. The server boundaries are dynamic in that self-balance can occur at run time. If one of the servers breaks down or a server process crashes, server boundaries can be adjusted to spread the load over the remaining servers. Most seamless servers use load balancing schemes rather than load sharing (load balancing constantly monitors and balances the load evenly among servers; load sharing is the static distribution of load in an attempt to avoid overloaded servers). The performance of the system may decrease with an increase in players under active load balancing because of overhead involved in constant monitoring [14]. Load balancing is implicit in seamless servers to a certain extent as clients are initially automatically connected to an appropriate server [11, 12].

BigWorld™ Pty Ltd. [13] claims that their technology based on seamless servers will revolutionize the gaming industry by removing the zone limitations of most MMOGs [13]. Thread synchronization is the key factor in this model: the game server is a single multiprocessor machine with multiple threads for partitioning the world. The server technology assigns and reassigns servers to game world areas based on CPU usage, constantly balancing server resource allocation to optimize coverage throughout the entire game world.

However, uncertainty always prevails in a seamless architecture [11]. Since threads take a great deal of overhead, synchronization may be difficult. Message passing techniques have to be implemented in order to avoid using potentially stale data. Asynchronous processes have a larger number of failures that have to be accounted for. Complications in world building, art, operations, and expansion appear to make the seamless server model not worth the effort involved. It can lead to a longer development schedule, increase the complexity of practically every line of game code, and impose limitations on what can be done in the game itself [11].

1.3 Network Issues

Considerable care must be given to the security, performance, and reliability of the data communications in a MMOG. Hacking of game packets with utility programs can be avoided by using bit-shifted data packets or encryption [12]. However, encryption is time-consuming and may be impractical, thus only key details (such as credit card information) would usually be encrypted. Data-streams should always be checked to guarantee authenticity.

Data packets from the player to the server should be as small as possible to minimize congestion at the server end. Tests by Farber [15] indicate that a maximum packet size of around 80 bytes is efficient.

Efficient data routing schemes can improve performance. Cybernet's real time intelligent routing technology, called OpenSkies™ [16], is based on nearly two decades of U.S. military research in the field of multi-player simulations. Their high level architecture (HLA) provides a scalable distributed network system for teleconferencing and other real-time interactions. Cybernet adapted HLA to create a system for massively multiplayer online games, and were able to reduce bandwidth load by 25% to 90% (tests used 500 clients, and packets were sent every 0.1 second [16]).

Data caching and buffering enhance performance by reducing the time taken to respond to requests. Data sent from the servers to the players may also be buffered to overcome latency due to differences in network speeds on the server and player sides. Cybernet uses caching in their real-time intelligent routing technology [16]. A distributed network running their software reduces the bandwidth to half. Intelligent routing also tracks a packet's source and destination, making it possible to consolidate streams.

Data reliability is an important concern of MMOGs (thousands of customers may have spent months or years building up their characters). If there are no back-ups, one would go out of business quickly. Reliability can be improved by the use of clustered caches (e.g., Coherence™ [17]) that store transient application data (databases are normally optimized to store transactional and persistent, not transient, data).

Duplicate servers can also provide fault tolerance; if one server goes down, a duplicate can help restore the system – including transient data – in case of a crash.. Load balancers such as BigIP [13] allow multiple computers to be hooked together so that they all appear to have the same IP address to the outside world. One machine is used as the primary system; if this goes down, BigIP automatically routes traffic to the next in line. Constant backups of the data allow restoration of the system in the case of faults.

One of the main design considerations in today's online gaming industry is the need for continuous upgrade of the game system and its features. To attract and retain customers, regular changes to the game with new features and scenarios are necessary. To cope with this demand, more servers may have to be added to the system. Upgrades have to be done while retaining player data and without interrupting their play. Hence, the basic design of the servers has to be modelled suitably to facilitate easy expansion of the system.

Server side updates are easily done in zone-based servers. On the client side, updates are done by providing patches for downloading from the game web links.

2. The CSP Game Model

2.1 Introduction

We present first a CSP model of the game. Successful construction of a verified model can greatly decrease the time and effort required to get the actual implementation running. This particular model is fairly straight-forward; a different version – far more complex – has been developed by McInnes [6].

Our online game system consists of several servers, and numerous clients (players) that connect to play the game over the network. The servers may be classified in terms of the specific tasks that they perform, e.g. the login server takes care of logins, the physics servers are involved in managing players' moves, and the database servers maintain player information.

Clients begin by connecting to the login server and receiving information on the game status; the login server will inform the physics servers that the player is allowed to join the game. The physics servers then handle all move requests generated by the players. The concurrency structure of this system is fairly straight-forward, and the only significant issue is verification that a dying player will not affect the play of those remaining.

The CSP model includes overwriting buffers ([e.g., 16]) between the output of the physics servers and the inputs of the players. Thus, if a player dies, these buffers will continue to swallow the outputs of the physics servers, allowing them to continue. In general, the size of the overwriting buffer (i.e., the number of messages it may store) must be set carefully [18]: to avoid losing important information, the buffer must be able to store the largest number of messages that the server may send before the server waits for a response from the player.

(There is another possibility of failure that we do not address: we have no mechanism to guarantee fair access of players to servers. Thus one – or more – players may turn around so fast that other players may be blocked indefinitely. This is a common problem, and does have a number of solutions; see, e.g., Andrews [23].)

2.2 *The System Model*

The basic model consists of a login server (`LS`) and a physics server (`PS`) (Figure 1). The login server connects to the login data base (`LDB`) and the physics server is linked to the physics data base (`PDB`). Players are labelled P1, P2, etc., and act as clients to the servers.

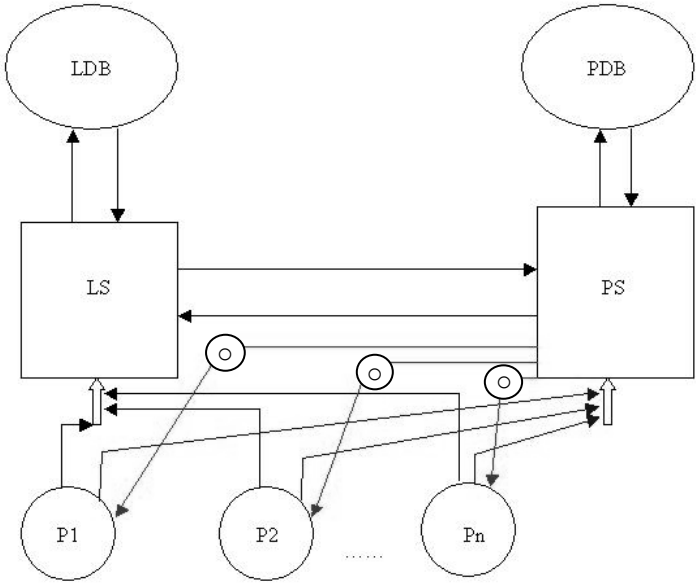

Figure 1. Basic CSP model of the game system. The broad arrows on the login server (`LS`) and the physics server (`PS`) represent Any-to-One (shared) channels. The circles labeled "`o`" are the overwriting buffers.

Players enter the game by making a request to the login server. The `LS` has a shared channel `login_chan` which is used by all players (this channel is indexed in CSP to model the sharing). The login process from the perspective of a player is described in CSP by the first two events of the `PLAYER(i)` process, as shown below. (The `mov` branch will be explained later.)

```
PLAYER(i)   =  login_chan!i -> ack?i -> PLAYER1(i)

PLAYER1(i) =  (mov!i -> response.i?result-> PLAYER1(i))
              []
              (quit!i -> PLAYER(i))
```

The input channels (`response`) of the players are renamed `OWBtoPlayer`, and the outputs of the overwriting buffers are connected to these new channels; the response channel is assigned to the input of the overwriting buffers, thus making the insertion of the buffer transparent to the physics server:

```
BufPLAYER(i, BufDepth)  =  OWB(BufDepth, response.i, OWBtoPlayer.i)
                           [|{| OWBtoPlayer.i |}|]
                           PLAYER(i)[[response.i <- OWBtoPlayer.i]]
```

The login server waits for players' requests on the channel `login_chan`, checks with the login database, informs the physics server that the player has registered, and waits for an acknowledgement for the physics server; the login server then returns to its initial state. The CSP description of the login procedure is the process `Login_SERVER` (note that, at this level of abstraction, no real check is done on the validity of the player's request, and we model the check and response as one event):

```
Login_SERVER  =  login_chan?i:PLAYERS -> check1_db!i ->
                 LgnToPhy!i -> ack?i -> Login_SERVER
```

The physics server is always waiting either for a registration notification from the login server or for a move (`mov`) request (over another shared channel) from a player. An external choice (the `[]`operator) over the `LgnToPhy` and `mov` channels is used to represent this behavior in CSP. If a login is received on `LgnToPhy`, an acknowledgement `ack!i` is sent back to the login server. If a move request from the player is received on the shared channel `mov`, the PS checks the database (`check2_db!i`) to see if the move is legitimate, updates the database if it is (`mov_update`), informs all players of the move just made (the `UpdateAll` process) via the `response` channels to the overwriting buffers, then repeats. (Note that, to keep the model simple, the physics model does not include verification of player's registration.)

```
physics_SERVER  =  (LgnToPhy?i -> ack!i -> physics_SERVER)
                   []
                   (mov?i:PLAYERS -> check2_db!i ->
                    mov_update -> UpdateAll -> physics_SERVER)
```

The server and player models are connected to make up the entire system. The server system (SERVERSYS) is defined by first linking the databases with their respective servers via the appropriate channels (check1_db and check2_db) and then joining the two subsystems, via the channel LgnToPhy, to create SERVERSYS. The complete system (SYSTEM) is then constructed by synchronizing SERVERSYS via the appropriate channels with the interleaved composition of the players (ALLPLAYERS). (Note that several of the internal channels are actually hidden when the verification is run.).

```
LGNSYS     = Login_SERVER [|{|check1_db|}|] Login_dbase_SERVER
PHYSYS     = (physics_SERVER [| {| check2_db|} |] Mov_dbase_SERVER)
SERVERSYS  = (LGNSYS [| {| LgnToPhy, ack |} |] PHYSYS)
ALLPLAYERS = |||i:PLAYERS@(BufPLAYER(i, depth))
SYSTEM     = ALLPLAYERS[|{|login_chan, mov, response|}|]SERVERSYS
```

2.3 The Specification

The major verification goals are absence of deadlock and livelock and the confirmation that one (or more) dying players will not block communication with the remaining players. The specification thus simply consists of a single player that does not fail. We set the depth of the overwriting buffers at one, since this is sufficient to demonstrate that a failed player will not block the servers. (We are more liberal in the Java implementation below.)

The implementations to be tested against this specification are systems of two and three players, one of which plays correctly as defined above (`PLAYER(i)`). The other

players are modified so that they terminate (via `SKIP`) immediately after executing `login_chan!i`; this is sufficient to cause the Physics Servers to attempt to send them messages – which will not be accepted. These two versions (with appropriate hiding) should refine the specification.

2.4 Verification

The FDR tool verified that systems of one, two, and three working players were free of deadlock and livelock, and that the two systems including the terminating players did refine the working single player specification. The overwriting buffers are thus successful at absorbing messages that can not be delivered.

3. Implementation in JCSP.net

3.1 JCSP.net

JCSP is a library that provides facilities to implement channel-based CSP systems in Java [9, 10]; channels are initialized to have an "output end" at the sender and an "input end" at the receiver, respectively. The JCSP.net version includes the option of setting up channels over the network by using a centralized channel name server (CNS). Channels connecting processes located on different machines (or invoked independently on one machine) must be registered at the CNS. Channel ends can be moved around a network, providing a great deal of flexibility and the possibility making major changes in a system dynamically.

Early experiments [10] on cluster systems indicated that JCSP.net could outperform similar toolsets such as *mpiJava* and IBM's *TSpaces*. Distributed robust annealing (a JCSP.net version of [18]) and real-time systems [20] monitoring movement of pedestrians have also been successfully demonstrated. JCSP.net should be ideal for game systems.

3.2 JCSP.net Implementation of the Servers

3.2.1 Login and Physics Servers

The structure of the game server system during the startup phase is shown in Figure 2. The system consists of the Login Server, the Login Database Server, and one or more combinations of a Physics Server and its associated database. Each Physics Server is associated with a particular game domain. Initially, players do not know the name of the channel via which they will communicate with their physics server.

The Login Server has an Any-to-One channel, registered at the CNS, over which players (knowing the channel name) register to initiate play. When a player registers it will send along the name of the channel (dashed) over which it will listen for the Physics Server. The Login Server will then pass this channel and the player's request to the Physics Server.

The Database Server connected to the Login Server is used to check the player's ID; if successful, the login is validated. If the player is starting a new game, a random location in the game domain is assigned to the player; otherwise, a player will be placed at his last location. The Physics Server is informed of the player's ID, its game location, the channel over which the player will listen, and possibly other information about the player's status. The Player is sent the same information, along with the name of the shared (anonymous) channel that will be used by the player to communicate with the Physics Server. The login server is then ready to accept login requests from other players.

The Physics Servers are associated with specific geographical zones in the game world, and each has its own database. These servers can be distributed over the network on different machines. Figure 3 shows the configuration of a Physics Server with its database after the players have connected over the shared channel. Overwriting buffers (the bubbles marked O in Figures 2 and 3) are used on the channels from the physics servers to the players. As noted above, the buffers maintain data flow if the network is slow on a player's side, and allows a server to continue if a player ceases to accept messages.

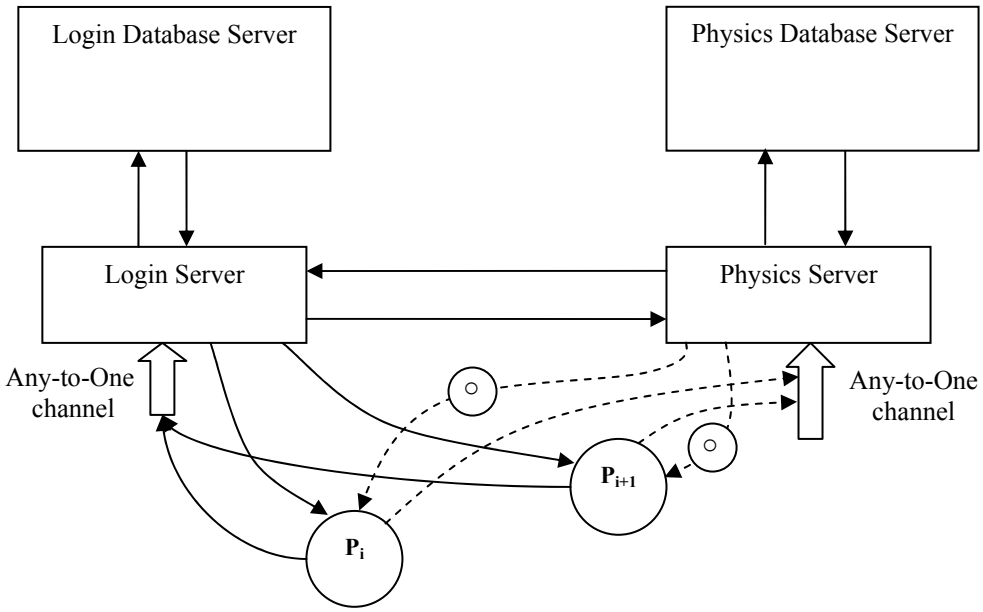

Figure 2. At startup, Players request a specific Physics Server from the Login Server and submit the channel over which they will listen via an overwriting buffer; Players are then informed of the appropriate channel to connect to the desired Physics Server.

The code fragments below show the creation of a shared (anonymous) channel by a Physics Server, the determination of the location of that channel by the Login Server, the transmission of the location from the Login Server to its reception by the Player, and the assignment of that location to a NetChannelOutput at the player. toPlayer and fromServer are the write and read ends of the channel from the Login Server to the Player; playerOut is the write end of the channel from the Player to the Physics Server.

```
// Physics server:
      NetAltingChannelInput anonChannel =
            NetChannelEnd.createNet2One();

// Login server:
      NetChannelLocation serverLocation =
            anonChannel.getChannelLocation();
      toPlayer.write(serverLocation);


// Player:
      NetChannelLocation serverLocation =
            (NetChannelLocation)fromServer.read();
      NetChannelOutput playerOut =
            NetChannelEnd.createOne2Net(serverLocation);
```

During play, each Physics Server receives move requests from players, updates its corresponding database server accordingly, and informs (as necessary) players in its zone of other players' moves. If a player moves from one zone to another, the player data in the previous server is moved to the new server to which the player now belongs. This movement is easily implemented using the mobile channel ends of JCSP.net.

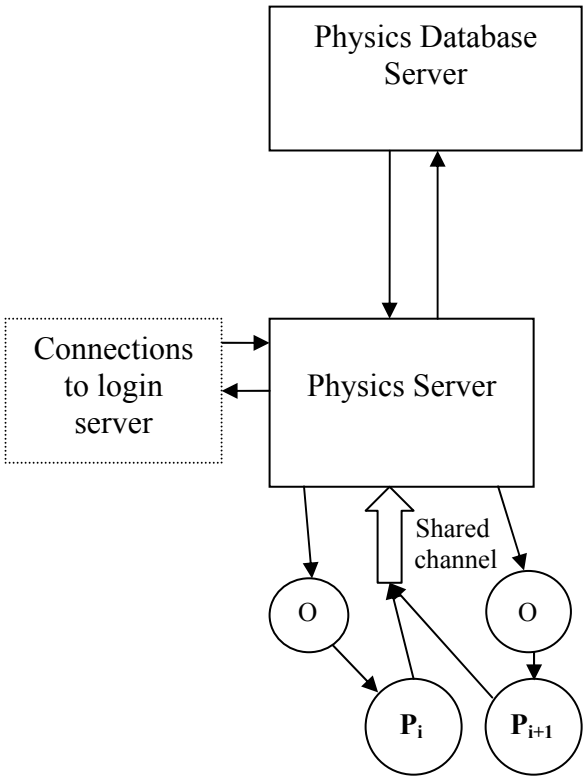

Figure 3. A Physics Server with players connected; overwriting buffers (O) between the Physics Server and the players protect the server if a player is either slow at accepting or ceases to accept incoming messages.

We stated above that the size of an overwriting buffer should be chosen to accommodate (at least) the number of messages one process can send to a second before waiting for a reply from the second since, if the buffer is too small, data could be overwritten before the second process fetches it [18]. In this application, however, there is no limit on the number of messages the server can send to a player, as the moves of other players may generate an unlimited amount of traffic. We have arbitrarily chosen a buffer of size four in our tests. (In practice, if packets are overwritten, the buffer could be expanded on the fly. Adding sequence numbers to packets would allow detection of overwriting.)

We use the JCSP class OverWritingBuffer to create a channel end with an OWB of size four; the appropriate code is:

```
NetChannelInput Server2Player =
      NetChannelEnd.createNet2One (new OverWritingBuffer(4));
```

3.2.2 Database Servers

There is one database server connected to the Login Server and one for each of the Physics Servers. The login database, as noted above, keeps track of registered players. The physics databases store information on the location and status of the players and inanimate objects

(game items: treasures, etc.) in each zone. The game items, randomly placed at startup, can be picked up by players. A Java hashtable [21] is used to maintain the player databases.

3.3 JCSP.net Player Model

The Player starts up by first creating a channel end to connect to the shared input channel on the Login Server. It next creates another channel end on which it will receive messages from the physics server. The Player then logs in, sending its user ID and the physics channel end to the Login Server. The Login Server will pass 1) the required player information (including the physics channel end received from the player) to the appropriate Physics Server, and 2) the physics channel end to communicate with the Physics Server to the player. The Physics Server will then connect with the player over the received channel, sending the Player the shared channel end to talk back to the Physics Server. The new player receives all the information about items present and other players on the board. (Note that the Player need not have information about the location of physics servers; this information is hidden from players, thus providing some server security.)

The player-side (non-human) model is divided into three processes – keyboard, screen, and graphics. The keyboard generates random moves and sends them to the physics server (randomly dispersed in time to model an actual player; the random delay has been set to be between 2 and 5 seconds). The screen receives updates of the moves. The graphics section is connected to the screen and takes care of refreshing the board on the player side. The graphics is based on the Java Swing API (the graphics were turned off in the performance tests below).

4. Performance Analysis

4.1 *Previous Work on Performance Measures*

Several groups have recently studied multiplayer game performance. Distributed server systems have been designed and analyzed to measure the traffic behaviour over the network. The major concern with networked traffic is *transmission delay* or *latency*. Other issues such as bandwidth requirements, fault tolerance, and expansion of the system to allow more subscribers are also discussed.

4.1.1 Network Transmission Delay (Latency)

Network transmission delay or latency of the network is defined as the time taken for the packet sent from the sender to reach the receiver. The main contributions to latency are the performance of the transmission medium (optical fiber, wireless, etc.), the size of the packet, router hardware, and other processing steps at the time of transmission.

Farber [15] describes the traffic scenario for an online game system with 30 players. Their tests consisted of a client-server model to evaluate the fast action multiplayer game "Counter Strike" played over a LAN. Each player lasted 30 to 90 minutes and the traffic was observed for 6.5 hours. Client and server traffic were studied separately. The player packet size was around 80 bytes. Tests showed that a round trip time of 50-100 milliseconds is sufficient for fast play games.

4.1.2 Bandwidth Requirements

Pellegrino et al. [22] have described a procedure to calculate the bandwidth requirements, both on the player and server sides. They note that the client-server architecture is not

scalable as it requires large bandwidth, but overhead on the player side is the drawback in peer-to-peer architecture. A model combining the merits of the two architectures is proposed to have the lowest bandwidth requirement.

Their tests had four players playing BZFlag, an open-source game, on Pentium-III based PCs running Redhat Linux 7.0 connected via a Fast Ethernet Switch. The player update period, $T_{U,}$ referred at the player side, is defined as the time taken for the player to make a move, the move to reach the server, the server to update the database and all players, and the server to send an acknowledgement to the player; the random delay between two consecutive moves is also included. The number of bytes sent by the player is L_U. The upstream bandwidth at the client side is the bandwidth required for every move, i.e. L_U/T_U. The client downstream bandwidth includes updates received from the player for each move made by other players, i.e. $N(L_U/T_U)$. Thus,

$$\begin{aligned} \textit{Total Client Bandwidth} &= \textit{Client Upstream Bandwidth} + \textit{Client Downstream Bandwidth} \\ &= (N+1)\, L_U/T_U . \end{aligned}$$

Similarly, the total server bandwidth is $N(N+1)L_U/T_U$, indicating that it scales quadratically.

4.2 Simulation Setup

Our tests were done by running the entire game system on Windows and on Linux; we also did a smaller test using Linux and Windows at the same time. There was no substantial difference in the performance on the different operating systems.

For the primary Windows tests we set up a game board with a playing field of 1,000 × 1,000 cells. This field is split into five zones of 200 × 1,000 cells, with each zone managed by one Physics Server. The structure of this world is shown in Figure 4.

We place the Login Server on one machine and the Physics Servers on five additional machines. The players are allocated to another 25 machines: tests were run with 5, 10, 20, 50, 100, 200, 400, 700, and 1,000 players. The performances of the servers, the players, and the response times for both the servers and players were recorded. The machines were Pentium-IV (3 GHz) PCs running Windows XP with 1 GB of RAM each; they were on a 100 Mbps connection.

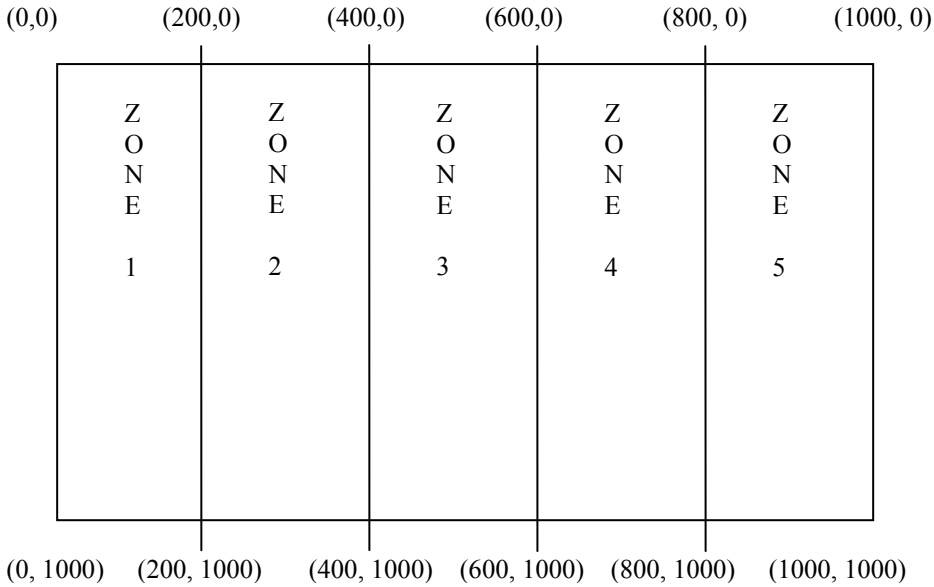

Figure 4. Game board with five zones, each managed by one physics server.

4.3 Performance Results

The player end-to-end response times were measured for a number of players running from 100 (10 players on each of 10 player machines) to 1,000 (40 on 25). The flow involved in calculating player end-to-end time is shown in Figure 5. The Java *date* class was used for measuring time.

Table 1 shows the detailed results for each of the five physics servers. The player end-to-end response time with 100 players is about 16 milliseconds. The response time increases to about 65 ms with 1,000 players; this is well within the acceptable value of 100 ms. Figure 6 shows the average response times over all physics servers for 100, 300, 500, 700, and 1,000 players. An increase of players by a factor of 10 yields a factor of only 4 in the response time.

The bandwidth required, both at the player and server side, is calculated using the equations provided by Pellegrino [22]. The average time between player moves, $T_{U,}$ is 3.5 seconds. The maximum value of N is assumed to be 200 as there are five servers each taking around 200 players.

T_U = 3.5 seconds,
L_U = 80 bytes = 80 * 8 = 640 bits
Client upstream bandwidth = 640/3.5 = 183 bits/second
Client downstream bandwidth = 200 * 640/3.5 = 36,572 bits/second
Total client bandwidth = 36,755 bits/second
Total server bandwidth = N * (N+1) L_U/T_U = 200* 201*183 = 7.36 Mb/sec.

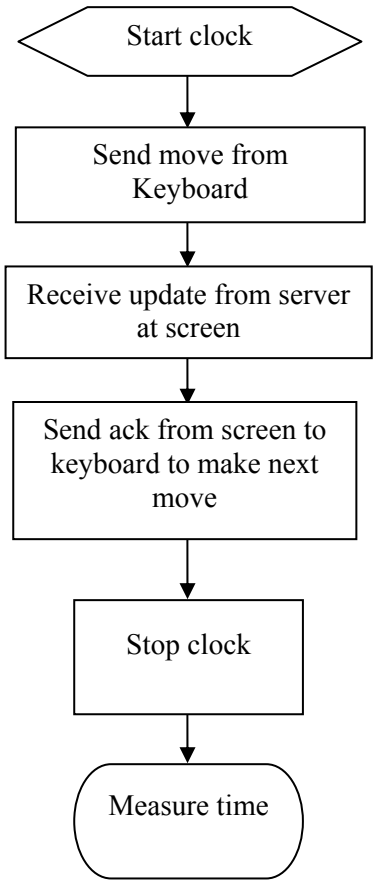

Figure 5. Flow chart to calculate response times.

Table 1. Player End-to-End Response Time

Players	Average player end-to-end time (in ms)	Player end-to-end response times for servers 1 to 5 (players:time in ms)					Machines × players
		1	2	3	4	5	
100	16	20: 16	17:16	23:16	18:16	22:16	10 × 10
200	16	39:16	33:16	42:15	43:16	43:16	10 × 20
300	16	51:16	55:16	68:16	68:16	60:16	15 × 20
400	25	64:16	78:16	93:32	81:31	84:31	20 × 20
500	28	80:31	97:16	116:31	92:31	115:31	25 × 20
600	41	96:32	126:32	137:47	108:47	133:47	20 × 20, 5 × 40
700	47	122:47	135:47	167:62	122:47	154:47	15 × 20, 10 × 40
800	57	133:47	156:47	180:63	155:63	176:63	10 × 20, 15 × 40
900	60	149:47	174:63	208:63	167:63	202:63	5 × 20, 20 × 40
1,000	66	166:63	189:63	223:63	192:63	230:78	25 × 40

The bandwidth at the player side and server side are thus about 36.8 kbps and 7.36 Mbps, respectively (this includes TCP/IP header information). With 40 players per machine, the bandwidth required would be about 1.5 Mbps. These values are well within the 100 Mbps of the speed of this network.

The response time can be improved by adding more physics servers to share the load. This, in turn, would give better player end-to-end response time.

Because each player machine in the test had 40 players, the network bandwidth and the hardware resources at each machine affected the player-side timing. With fewer players per machine, the player-side response time should improve as hardware and network performance would be better. In Table 1, the response time up to 500 players is low as the maximum number of players in each machine is 20. Once we had 40 players per machine, the increase in response time was steeper. This jump was seen when the number of players was increased from 500 to 600, a point where five machines were loaded with 40 players. Moving players between zones did not appear to add significant overhead.

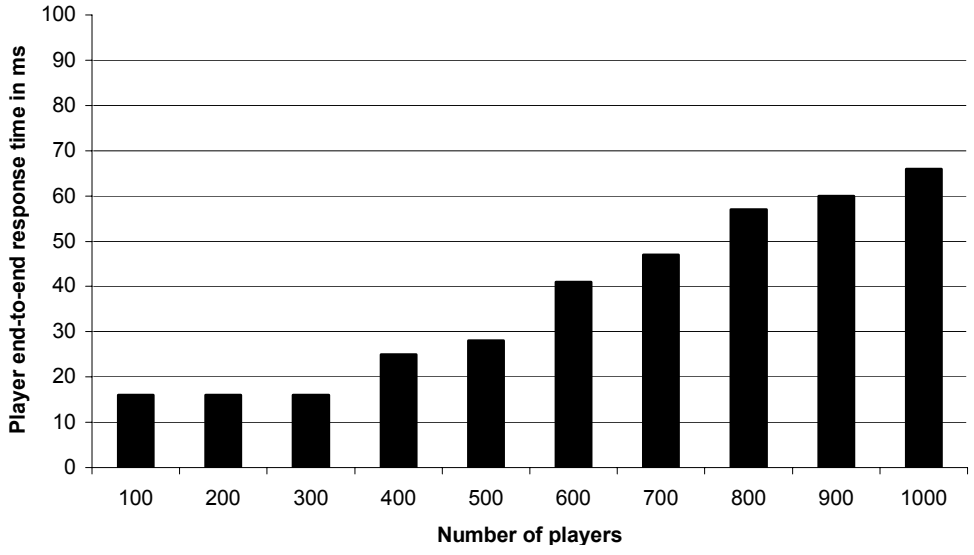

Figure 6. Average player end-to-end response time vs. number of players.

The CPU load on the machines running 40 players was nearly 90%. On the serverside, the CPU load was around 85%; this is within the industry standard, but is higher than recommended (70%).

A much smaller test was run on 10 Linux machines running Suse 9. These machines were not identical to the Windows machines and detailed measurements were not made. No problems were encountered, however, and the performance appeared comparable.

4.4 Fault Tolerance

We have run a limited number of tests for fault tolerance. As described above, we explicitly added features (the overwriting buffers) so that a failure to respond on the part of one or more players would not prevent other players from continuing. This was tested by disconnecting active players; the system continued to service properly the remaining players. We were also able to shut down one of the physics servers with no effect on the rest of the system (players on the system shut down would lose any state not yet transferred to the physics database and, in the absence of overwriting buffers between the login and physics servers, a login server attempting to contact a dead physics server would stall.).

4.5 Summary

The overall performance of our system matches the industry standards of online games. The player delay is in the range of 16 to 80 milliseconds, which is acceptable for fast action games such as racing or combat. Online games that have comparatively slow action, such as role-playing games, can tolerate delays of up to 150 milliseconds and would have a larger margin.

5. Summary and Future Work

The project began with a detailed study of online gaming. A simple game system was then designed in CSP and verified to be deadlock-free using FDR. The next stage of the project was to implement the design in Java using JCSP.net. Tests on both Linux and Windows were successful. The final results showed that a system of five servers and 1,000 players on 25 machines met our performance goals and the timing requirements of online games.

The project also showed that a robust gaming system with a significant amount of concurrency could be efficiently developed using a CSP-based approach. For game developers, this project demonstrates the facilities of JCSP.net and introduces them to game design using a formal software engineering methods and appropriate verification tools. This approach should lead toward better verification and testing before releasing the game in the market, and decreased development time.

This game system can easily be improved and modified to suit the latest trends. More game features, such as community development and instant chat, could easily be added; the chat feature would be particularly easy to implement using JCSP.net's mobile channels (we assume that players are not behind firewalls). On the server-side, servers could easily be added to support more players. This can be done dynamically in JCSP.net. Standard fault-tolerance techniques based on additional back-up servers could be implemented to make the system more robust. The game should be easily implemented in occam-π.net and C++CSP.net, two other systems based on CSP.

The hash table used on the servers storing player data is based in the primary memory; it requires RAM of 1GB and a good processor (Pentium-IV). This can be modified to store

the data on a hard disk, which would allow access when the server is shut down during maintenance or crashes. We would, however, see a decrease in performance.

A more robust system could employ commercial databases such as the Oracle Database 10*g*. Commercial databases have self-managing facilities, which help to maintain the data during faults or crashes.

Acknowledgements

Many thanks to Allan McInnes for his thorough preliminary review of this paper, and to the very conscientious CPA 2006 reviewers.

References

[1] JCSP.net Home Page. http://www.jcsp.org/
[2] J. C. McEachen, "Traffic analysis of internet-based multiplayer online games from a client perspective," in *ICICS-PCM,* IEEE*,* vol. 3, pp. 1722-1726, Singapore, 2003.
[3] J. Krikke, "South Korea beats the world in broadband gaming," IEEE Multimedia, vol. 10, Issue 2, pp. 12-14, April/June 2003.
[4] T. Nguyen, B. Doung, and S. Zhou, "A dynamic load sharing algorithm for massively multiplayer online games," in *11th IEEE International Conference on Networking*. pp. 131-136, 2003.
[5] *Zona Inc.*: http://www.zona.net/company/business.html.
[6] McInnes, Allan I. S., "Design and Implementation of a Proof-of-Concept MMORPG Using CSP and occam-π", in Proc. 2005 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'2005), vol. 1, pp. 194-200, CSREA Press, Athens, Ga., ed. H. R. Arabnia, Las Vegas, CSREA Press (USA), June, 2005.
[7] A. W. Roscoe, "The Theory and Practice of Concurrency," Prentice Hall Series in Computer Sciences, ed. R.B. C.A.R Hoare. Hetfordshire: Prentice Hall Europe, 1998.
[8] F.S.E. Ltd., *Formal Systems Software*: http://www.fsel.com/software.html
[9] Quickstone Technologies Ltd.: http://www.quickstone.com/xcsp
[10] P. H. Welch and B. Vinter, "Cluster Computing and JCSP Networking," in *Communicating Process Architectures 2002,* pp. 203-222, IOS Press, Amsterdam.
[11] T. Alexander, "Massively multiplayer game development," Charles River Media, Inc. Massachusetts, 2003.
[12] T. Barron, "Multiplayer Game Programming," Prima Publishing, Roseville, 2001.
[13] BigWorldTech: http://www.bigworldtech.com.
[14] D. Bauer, S. Rooney and P. Scotton, "Network infrastructure for massively distributed games," *1st Workshop on Network and system support for games*, ACM Press, Bruanschweig, Germany, pp. 36-43, New York, 2002.
[15] J. Farber, "Network Game Traffic Modelling," University of Stuttgart, Stuttgart, Germany, 2002.
[16] Cybernet's OpenSkies: Networking engineer - Introduction: http://www.openskies.net/files/Introduction.pdf.
[17] "Reasons to use Coherence™: Increase application reliability": http://www.tangosol.com/coherence-uses-a.jsp
[18] G. S. Stiles, "An occam-pi Implementation of a Verified Distributed Robust Annealing Algorithm," in Proc. 2005 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'2005), vol. 1, pp. 208-218, CSREA Press, Athens, Ga., ed. H. R. Arabnia, Las Vegas, CSREA Press (USA), June, 2005.
[19] University of Kent at Canterbury: http://www.cs.kent.ac.uk/projects/ofa/jcsp/
[20] S. Clayton and J. M. Kerridge, "Active Serial Port: A Component for JCSP.Net Embedded Systems," in *Communicating Process Architectures 2004,* pp. 85-98, IOS Press, Amsterdam.
[21] Java Documentation: http://java.sun.com/j2se/1.3/docs/api/java/util/Hashtable.html
[22] J. D. Pellegrino and C. Dovrolis, "Bandwidth requirement and state consistency in three multiplayer game architectures," NetGames, May 22-23, 2003.
[23] Gregory R. Andrews: "Foundations of Multithreaded, Parallel, and Distributed Programming", Addison Wesley, 2000.