# Robust Networking in Multiplayer Games

Mihail Dumitrescu
California Institute of Technology
1200 E.California Blvd
Pasadena, USA
mihaid@caltech.edu

Nihar Sharma
California Institute of Technology
1200 E.California Blvd
Pasadena, USA
nihar@caltech.edu

## ABSTRACT

This paper documents the development of a robust, scalable networking infrastructure for a multiplayer game developed using Pygame [1] and PyOpenGL [2] in the Python programming language.

The work undertaken by the authors involves developing a networking library from the ground up for an open source multiplayer game called Spacewar [3] aimed at eliminating any single point of failure vulnerability from the client-server architecture along with adding various enhancements to the performance of the game. The problem of single point of failure vulnerability is tackled by designing the library as a unification of client and server instances that can choose to perform the operations of either based on the requirements of the game. The performance enhancements include the incorporation of numerous modern techniques in networked game programming including client-side prediction algorithms, server-side computational buffers and host-switch smoothing algorithms.

## Categories and Subject Descriptors

C.2 [**Computer-Communication Networks**]: Network architecture and design, Distributed networks

## General Terms

Algorithms, Design, Performance

## Keywords

Multiplayer games, Distributed networks, Client-server

## 1. BACKGROUND

As internet connectivity ceases to be a limitation, more and more games are starting to take advantage of the ”social” experience in gaming by letting users play with their friends online. It is not hard to see a world where a virtual network between friends would be persistent even past game sessions [4] [5] [6]. But before this ”virtual presence” can become a reality, games will remain the most popular social interaction medium for friends online. To manage games on this kind of scale, several network architectures have been experimented with over the years.

Peer-to-Peer (P2P) architectures explored all nodes of a network locally managing the game state [7]. This approach variably employed fully connected or overlay networks depending on the requirement of the game and other considerations. The advantages here are clear - there is no single point of failure, so the network is robust in that sense but also flawed since it is harder to stop cheating as the messages are not being verified by any one server [8] [9]. Look-ahead cheating is a method of cheating within a peer-to-peer multiplayer gaming architecture where the cheating client gains an unfair advantage by delaying his actions to see what other players do before announcing its own action. A client can cheat using this method by acting as if it is suffering from high latency; the outgoing packet is forged by attaching a time-stamp that is prior to the actual moment the packet is sent, thereby fooling other clients into thinking that the action was sent at the correct time, but was delayed in arrival. A partial solution is the Lockstep protocol [10]. Furthermore synchronization of the distributed game state is difficult. This makes it a requirement to have a lobby for players to join, from which the game starts and foils many game designers' attempts at allowing new players to join mid-game. A good example of game employing a P2P architecture is Age of Empires.

The client-server architecture was ushered into multiplayer games with the advent of Quake. One central machine was designated as the ”server” and it was responsible for all the gameplay decisions. The clients simply sent their actions to the server, received the game objects that they should render and displayed them on the screen. The responsibility of this game evolution being fair, correct and responsive belongs to the server [11]. This model was both scalable and flexible, save for the server load bottleneck, since now clients could connect to a server and join an existing game, which was not possible in P2P architectures.

The client-server model was further improved when games started to introduce client-side prediction and simulation into the mix. The servers now sent some additional information about the state and trajectories of game objects so that clients could simulate some of the game physics on their own, reducing the bandwidth usage and increasing the amount of detail that they were able to render at the same time. This technique is still limited in capability for games that are becoming increasingly non-deterministic when considering the number of variables that client-action outcome depends on.

## 2. APPROACH

There were many considerations to be made when choosing the game for which we wanted to develop a networking library. Although the goal was to produce a generic library that can easily be reused by game programmers, we needed to handpick the game we would use to test this library with carefully. Firstly, the game had to be using a client-server network topology in its multiplayer setup. Secondly, we wanted the game to be deterministic in the computations involved for its game state since that would allow us to see the advantages of certain techniques like client-side prediction relatively easily. Also, we wanted the game to be written using certain techniques like client-side prediction relatively easily. Also, we wanted the game to be written using a library that was cross-platform, and relatively high-level so that game programmers could use our code in real-time game development without knowing the low-level mechanics of a C-like programming language. This is why we chose Spacewar, which was written in Python using both Pygame and PyOpenGL libraries. Spacewar is a multiplayer head-to-head death match (DM) game and there are several reasons why this is important. DM games as opposed to their massively multiplayer online (MMO) counterparts like World of Warcraft, rely on rapid interactions and fast-paced gameplay. These games have short-lived highly variable game states. Any client can host an instance of the game, which is a quick session of rapid decision making. These are the type of games we wanted to direct our networking library towards since they are better served by the client-server architecture. MMO games have long-lasting game sessions in a persistent game world that involve dynamic social interactions of some kind. These have their own set of robust networking challenges which we will not explore here.

There are several important design decisions that needed to be made once the game codebase was examined. It was evident from the poor documentation and unconventional programming style that a lot of game programming was going to be unavoidable during the course of this project. The first couple of weeks were spent studying the codebase and cleaning it up. It was decided that we would rewrite the game logic to separate out the graphics rendering code from the game physics. This was an important step in establishing a separation-of-concerns guideline in our code so that we could exploit code reuse in the future. We decided to introduce a clean action-state model in the code, where chief operations would involve a get_actions and broadcast_update method for the server and a get_update and send_actions method for the client. Quite evidently, this allows the game code to function in the manner shown in Fig [1].

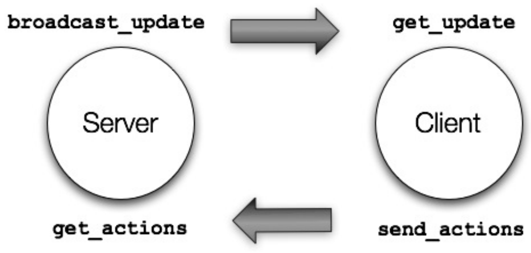

**Figure 1: Game Network Design**

The networking library was also planned to allow a user to simply indicate a server flag when invoking the game which would then enable the user to host a game session and distribute his IP address for other clients to connect to (who, in turn would use the invocation with only the host's IP address). Internally, the server flag would determine which operations a game instance would perform, which would affect socket functionality, packet sending/retrieval and ordering etc.

This brings us to our next design consideration which is the choice of protocol for implementation [12] [13]. Transmission control protocol or TCP certainly has desirable features for networking - guaranteed, reliable and ordered packets, automatic packet structuring for transmission and good flow control amongst others. On the other hand, User Datagram Protocol or UDP has none of these - no reliability or ordering of packets i.e packets may be out of order, duplicated or lost, no packet loss handling, everything needs to be handled explicitly. The only feature that UDP does promise is that if your packet arrives at its destination, it will arrive in whole or not at all. Although after these considerations it may seem that TCP is the logical choice, it would be a mistake to assume that. The reason for this lies in the way TCP implements the features it boasts. Unlike, web browsers or email etc, multiplayer games have a requirement for real-time packet delivery i.e most recent data is most valuable. So if a packet was lost, TCP would wait for that packet to be resent since it needs to guarantee delivery, but while it's waiting, either the client is not getting updates on the game state sent to it by the server or the server is not receiving newer client actions. Either way, the entire multiplayer game will break down and come to a halt till the packet which was lost arrives and then the packets queued up behind it start flowing. Even then, the packets being received are old now and will need to be processed in a single game tick causing a "jump" in the game. This is one of the reasons why UDP is the better choice for multiplayer games and we decided to implement our own mechanism to ensure ordering and account for packet loss in the library.

Initially, we also wanted to further develop our library into a distributed massively multiplayer networking library [14] [15] within the project timeline but this idea was later abandoned when several performance considerations that needed to be addressed in the client server model later proved non-trivial and time consuming.

### 2.1 Modularity in the game code

The most important step we took towards our goal is that of modularizing the game code. In other words, we separated the game physics code from the networking code and isolated the OpenGL graphics code from the game state calculations. This process was time consuming owing to the poor documentation of the initial codebase. But, once achieved, it allowed a smoother (as smooth as possible) transition from the current architecture to the redundant server model.

In order to be able to eliminate the single point of failure in the network, one important step was to allow clients to connect to a game instance not just directly to the server hosting the game, but also through any client that is currently playing in that instance. This will facilitate the propagation of a server address that the client sockets can use to maintain a redundant server among them.

### 2.2 AI

In addition to this, we have setup AI algorithms that can spawn players in a game and "play". These AI's are meant to simulate client behavior and allow us to see breaks in the game physics as we continue to modify the networking code.

Currently the AI's constantly shoot at players in the game and use euclidean distance path finding to move towards their opponent randomly. There is no further need to enhance their strategic algorithms for the purposes of this project. Although, we have made their targeting very accurate by allowing them to shoot at the precise locations of their targets to give the game more realism and challenge.

## 3. THE CLIENT SERVER MODEL

The game code has come a long way in providing us with a robust foundation to implement the new network architecture features that are our ultimate aim. Our first goal was to get rid of numerous bugs that made the gameplay experience extremely unstable. The game was initially not able to hold 3 players playing on the same local area network ( 2ms ping). We spent a great deal of time progressively refactoring and building upon a poorly commented/formatted codebase and were gradually able to structure the game and make the gameplay stable.

In the next phase, we completely rewrote the game handler and made it based on an action/state model. In this model, every client has certain actions that it sends to the server. In turn, the server handles the game physics using the actions from all the clients and then broadcasts the authoritative game state to every client connected to it. The network flow is of course of that shown in Fig [2].

## 4. CLIENT-SIDE PREDICTION

The client server model as stated above has limitations. On a LAN, because of the small 3ms lag, the game is responsive. However, such lags are very uncommon across the internet [16]. Once we experimented with 100ms or more artificially added lag, the game quickly became unplayable. This is due to the fact that every client action must travel to the server and the result must travel back before the client can see the results of his input [17]. In other words, there was

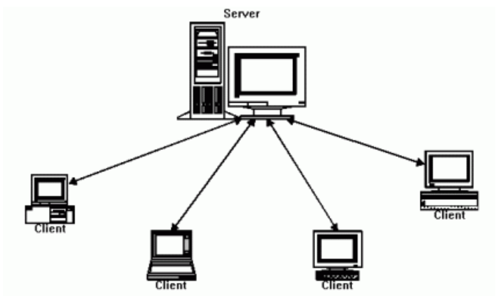

**Figure 2: Client-server model**

still a noticeable lag in the game response since every client action first needed to travel to the server and the server needed to calculate and respond with a new game state before that game state was displayed to the user. These delays can be anywhere from tenths of a second to a second over the internet. In the latter case, they render the game completely unplayable [18] [19].

To resolve this issue, we introduced the client-prediction technique mentioned earlier as soon as we receive user input. Now, instead of waiting for the users input to reach the server and the server to respond with a state, the client predicts/simulates the game on its own as soon as it receives some actions from the user. Since our game state can be approximated deterministically very well, this technique plays to our advantage. In other words, most of the time the state predicted by the client when user input in entered will be correct when compared to the servers simulation. So, when the client enters an input, it can begin rendering the results from that input immediately. Although the technique is conceptually straightforward to implement, the trivial implementation quickly runs into a lot of timing/lag issues. Since the severs game state is still authoritative, there are synchronization issues when the client and server disagree on the new game state. What happens is that the game starts to snap because the client jumps from its predicted game state into the servers game state. Figure 2 gives an example of such an instance if p represents a players position in a simple 2D map.

Our solution is detailed as follows. The now client keeps a base state (which is the last state received from the server), and predicts from it a simulated state which is then rendered. As above, let the client and the server start from the same state (p = (10, 10)). For the client, the simulated state (p*) and the base state are initially the same. Once the client does a few actions, the simulated state changes immediately to reflect it (p* = (12, 10)). When the server responds with the result from the first action the base state is updated (p = (11, 10)). The client now re-runs the whole simulation, taking into account only the actions which the server hasn't responded to yet (so p* = (12, 10), which is what we want).

```
# Replace with the latest received state.
del players, powerups
players = state['players']
powerups = state['powerups']
```

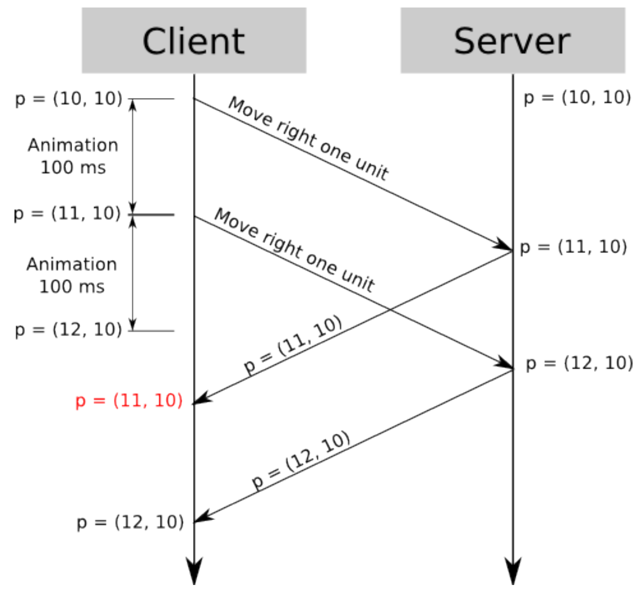

**Figure 3: Client and server state mismatch**

```
# Simulate up to our latest state.
for i in range(state['action_id'] + 1, action_id - 1):
  simulate_actions(prev_actions[i], players[args.name], dt)
  move_objects(dt, action_id <= i and args.graphics)
  collide_objects(action_id <= i and args.graphics)
```

Due to lag, packets from the server might arrive re-ordered. For this example, the response from 'action 20' could arrive before the responses to 'action 18' and 'action 19'. When the client processes 'action 20', it disregards both 'action 18' and 'action 19' when it does the simulation and from then on, will ignore the server responses to them. This works due to assumption that the server will process the actions from the user in the right order and without gaps. But as we stated earlier UDP doesn't work that way, so the following section explains how we (almost) guarantee that.

## 5. SERVER-SIDE BUFFERING

The server keeps a (conceptually FIFO) buffer of the messages (actions) that it receives from the clients. During a game tick, it pulls a message from the buffer and computes the next game state. This technique helps us with several protocol issues like packet reordering, loss and duplication. Since our buffer is indexed by the action ID, this maintains the ordering of the actions in the buffer automatically. In order to deal with the other two issues, we incorporated a redundancy into the message passing technique used by the clients. Each time clients send an action, they also send their previous two actions along with it. For instance, action tuples that might be sent in succession conceptually will be numbered as (3,2,1), (4,3,2), (5,4,3) etc. So this way, for the server to completely not receive action 3, all three of these packets must be lost in succession - unlikely! When the server does receives the messages, it puts them in the buffer, indexed based on their 'action id', overwriting if necessary. Thus, we are able to effectively deal with any packet loss or duplication. A more complete example would be to have the server buffer containing entries for action IDs of (10, 11, 12, 15, 16, 17), when the last action ID processed was 9. The point of the buffer is to keep rolling actions out at the same rate as others are added in. To reach this balance, we let the buffer grow (whenever there are buffer underflows, by 1) or shrink automatically (on overflows) within a range of approx 200ms, for a really good handling of connections that have up to $\pm$100ms of jitter.

The following code details how we manage buffer overflow, misses and underflow.

```
# Keep a buffer for each client address. Add to it only
# if we've not responded for that action id already.
try:
  if address not in self.handled or
    actions['id'] > self.handled[address]:
      self.buffer[address][actions['id']] = actions
except KeyError:
  self.buffer[address] = {actions['id']: actions}

...
# Get one action from the buffer for each address.
...
buffer = self.buffer[address]
ids = buffer.keys()
if len(ids) > 0:
  ids.sort()
  # Buffer overflow. Remove 2 actions from buffer.
  if len(ids) > BUFFER_SIZE:
    del buffer[ids.pop(0)]
    del buffer[ids.pop(0)]
    # Buffer miss. The action message from the client
    # did not arrive in time.
    elif address in self.handled and
        self.handled[address] != buffer[ids[0]]['id'] - 1:
      self.handled[address] = buffer[ids[0]]['id'] - 1
      continue
  data = buffer.pop(ids.pop(0))
  all_actions[address] = data

  self.changed.add(address)
  self.handled[address] = data['id']
else:
  # Buffer underflow.
  del self.buffer[address]
```

In order to reduce the bandwidth usage of the library, there was an important design alteration that needed to be made [20]. For instance, let's say we have 10 players, which includes 9 AIs and 1 human player (so the game is extremely active and has a lot of game objects being managed): we calculate that every state update takes an average of 3013 bytes (length of the string message since every character takes 1 byte). Similarly all the client actions that are processed by the server when computing an update take an average of 220 bytes. If the host sends the same game state to every user (as in the model detailed until now), the host would have to have a bandwidth of 3013 * 60 * 10 $\approx$ 1800 KBps for the server (and 180KBps download for the clients). Clearly, the library is worthless if it imposes a bandwidth requirement on this scale. The solution that we came up with for this problem was that instead of the server computing the next state and spoon-feeding it to the clients, it could instead tell the clients about the actions that it used to compute that state. This way the clients now perform incremental updates based on the actions they receive from the server. Since the actions are much smaller in size than the states, this would yield a marked improvement in bandwidth usage. As expected, when using actions for incremental updates, and sending one state update every 3 seconds (to insure that any de-synchronization due to corruption or temporary loss of connectivity is handled), the total bandwidth requirement becomes: ((221 * 179 + 3013) / 3) * 10 = 141 KBps upload for the server (and 14.1KBps download for the clients).

Due to the UDP protocol, it is clear that now our design poses the same in-order, without loss or duplication messaging requirements on the client side that the server tackled using the message buffer that we elaborated upon. So we use a client side buffer that performs nearly the same operations, but for the actions sent to it by the server. It is a bit different since the server also sends the full game state every 3 seconds, and these messaging methods have to be integrated.

## 6. CONNECTING THROUGH CLIENTS

Now that a lot of network performance issues were addressed, we started to work on the socket programming design that would allow a new client to connect to a game just by knowing the address of any other client currently in that game (i.e. the address of the host is not required). This is achieved by a special ”connect” message identification. When a new client connects, it sends this ”connect” message to the only address it knows is in the game (could be either another client or the host). If its a client, then it passes the address of the host to the client trying to connect and if its the host, then it send its own address for the client to store. This way every client in the game keeps the host address ready to be sent out to anyone to sends it a ”connect” message.

## 7. REDUNDANT SERVER MODEL

The architecture for the implementation of the host-switching mechanism involves keeping a redundant server address in the state of each player in the game. Currently, a backup server is being chosen at random by the current server. Once its chosen, the current server sends the address of the backup within its game state to all players in the game including the backup, who knows its chosen. Following this, all players realize that the current server is dead when it disconnects/leaves the game based on a timeout of death of approximately 0.5s. When this happens, all the players in the game connect to the backup server and it begins to perform the operations of the host by broadcasting its most recent game state. At this point, the new host chooses a new backup server and the process repeats on every server death. There are some performance issues with the technique as it is described above. Since the clients themselves don’t stop when a server dies (how can they - they don’t even know it happened!), the game state evolves for them past the last state broadcast by the server that died. Since the new server starts broadcasting from this state, the game jumps back to it and then resumes normal execution. This ”jump” reverts any actions made by the clients during that transitional period so any developments since are instantly rolled back to the back up servers (now actual servers) last known game state.

### 7.1 Smoothing the switch

Since the state revert-jump described above is naturally very undesirable for the players of the game, we keep a buffer with the backup server that stores the most recent actions of all the players so that when it is called into action by the death of the current server, it uses the actions stored in the buffer to ”smooth” the transition into the new game state it broadcasts.

## 8. TESTING

Game performance has been tested under simulations of widely varying latencies. Since game responsiveness cannot be quantified, some descriptions of the performance and the conditions being tested with are given below:

- The biggest advantage to a player is now that at any latency, the player never feels that his game is sluggish at all. This is a direct result of client-side prediction kicking in and letting the user see his or her actions rendered immediately instead of having to wait for the server to respond.
- High jitter conditions now have smooth gameplay. $(100 \pm 80)$ms of latency produce good results. Only the graphic of the passing missiles is slightly distorted due to frames getting dropped, but this is hardly noticeable during gameplay.
- High latencies of $(500\pm100)$ms are still playable. Although, due to the increased lag, players are not able to quickly see the rendering of missiles as they are shot by other players in the game. What is seen is something like that shown in Fig [4]

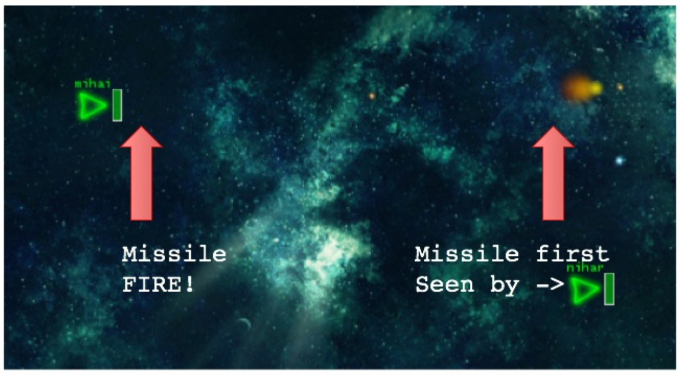

**Figure 4: High latency play**

- An integral part of testing was making sure the networking library does not hog bandwidth with the way that it manages game state. Testing the bandwidth usage with an increasing number of players in a game gave us results shown in Figure [5]. We can see that client performance scales really well with as number of players are increased but the server’s bandwidth usage scales almost quadratically, which is expected. Even so, this performance was not possible under the previous ”server-sends-state” design.

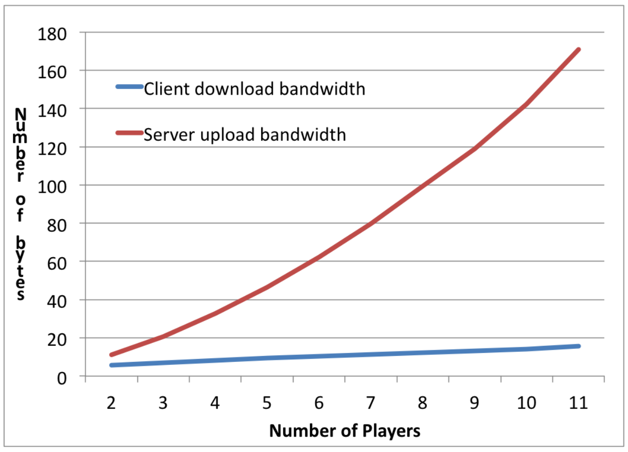

**Figure 5: Bandwidth Analysis**

## 9. SECURITY

Cheating, or better, cheating prevention plays an important role in online games. In our case, cheating is prevented only at a basic level, as a byproduct of our game design. Clients can only (directly) affect their own ship due to ways user request and are assigned an unique name. Furthermore, the construction of the action messages (and of their processing on the server side) allows only for a normal, controlled firing, movement and pause rate. However, these are the only constraints we can reasonably put on cheating. We cannot control how the user might abuse the communication protocol on the client side. This happens in mainstream games as well, however they use commercial anti-cheating software (like PunkBuster or Warden) that assures the game binaries are not modified. Our project is open source so it is easily modifiable, thus one can always program an AI to control the game for them. Also, once a user is a server (and any player might be one) the possibilities for cheating are many. They can change the source code to continue to respect the communication protocol but give themselves (or 'allies') advantages like a longer life or teleportation. However, in our case, this is easily noticeable.

## 10. CONCLUSIONS

Multiplayer games have started to receive an increasing amount of attention from the academic community in recent years. Tackling several requirements posed by such games has become a task much research is devoted to. For instance, several techniques have been proposed for the synchronization of game nodes (clients) in a distributed game network [21] [22] as this task of equal complexity with synchronization in classic real-time distributed systems. Many attempts have also been made to come up with optimal solutions to balance latency and fairness [11].

We have been able to encounter some of these issues at a smaller scale through the development of our networking library and hope that it will help game programmers everywhere easily incorporate robust multiplayer game play into their code.

## 11. REFERENCES

[1] Pygame, "Python api for game programming."

[2] PyOpenGL, "Python binding to opengl and related apis."

[3] I. Mallett, "Spacewar - a multiplayer space shooter."

[4] J. Smed, T. Kaukoranta, and H. Hakonen, "A review on networking and multiplayer computer games," in *IN MULTIPLAYER COMPUTER GAMES, PROC. INT. CONF. ON APPLICATION AND DEVELOPMENT OF COMPUTER GAMES IN THE 21ST CENTURY*, pp. 1–5, 2002.

[5] E. Castronova, "Network technology, markets, and the growth of synthetic worlds," in *Proceedings of the 2nd workshop on Network and system support for games*, NetGames '03, (New York, NY, USA), pp. 121–134, ACM, 2003.

[6] T. Manninen, "Virtual team interactions in networked multimedia games – case: "counter-strike" – multi-player . . .," in *IN PROCEEDINGS OF THE 4TH ANNUAL INTERNATIONAL WORKSHOP ON PRESENCE (PRESENCE 2001), PHILADEPHIA*, 2001.

[7] C. Neumann, N. Prigent, M. Varvello, and K. Suh, "Challenges in peer-to-peer gaming," *SIGCOMM Comput. Commun. Rev.*, vol. 37, pp. 79–82, January 2007.

[8] S. D. Webb and S. Soh, "Cheating in networked computer games: a review," in *Proceedings of the 2nd international conference on Digital interactive media in entertainment and arts*, DIMEA '07, (New York, NY, USA), pp. 105–112, ACM, 2007.

[9] J. Yan and B. Randell, "A systematic classification of cheating in online games," in *Proceedings of 4th ACM SIGCOMM workshop on Network and system support for games*, NetGames '05, (New York, NY, USA), pp. 1–9, ACM, 2005.

[10] H. Lee, E. Kozlowski, S. Lenker, and S. Jamin, "Multiplayer game cheating prevention with pipelined lockstep protocol.," in *IWEC'02*, pp. 31–39, 2002.

[11] J. Brun, F. Safaei, and P. Boustead, "Managing latency and fairness in networked games," *Commun. ACM*, vol. 49, pp. 46–51, November 2006.

[12] C.-M. Chen, T.-Y. Huang, K.-T. Chen, and P. Huang, "Quantifying the effect of content-based transport strategies for online role playing games," in *Proceedings of ACM NetGames 2008 (Poster)*, 2008.

[13] G. on Games, "Networking for physics programmers."

[14] A. Bharambe, "Colyseus: A distributed architecture for online multiplayer games," in *In Proc. Symposium on Networked Systems Design and Implementation (NSDI*, pp. 3–06, 2006.

[15] D. Bauer, S. Rooney, and P. Scotton, "Network infrastructure for massively distributed games," in *Proceedings of the 1st workshop on Network and system support for games*, NetGames '02, (New York, NY, USA), pp. 36–43, ACM, 2002.

[16] K. Claypool and L. Determines, "Latency and player actions in online games," *Commun. ACM*, vol. 49, p. 2006, 2006.

[17] D. Liang and P. Boustead, "Using local lag and timewarp to improve performance for real life multi-player online games," in *Proceedings of 5th ACM SIGCOMM workshop on Network and system support for games*, NetGames '06, (New York, NY, USA), ACM, 2006.

[18] S. Harcsik, A. Petlund, C. Griwodz, and P. Halvorsen, "Latency evaluation of networking mechanisms for game traffic," in *Proceedings of the 6th ACM SIGCOMM workshop on Network and system support for games*, NetGames '07, (New York, NY, USA), pp. 129–134, ACM, 2007.

[19] T. Henderson, "Latency and user behaviour on a multiplayer game server," in *Proceedings of the Third International COST264 Workshop on Networked Group Communication*, NGC '01, (London, UK, UK), pp. 1–13, Springer-Verlag, 2001.

[20] J. D. Pellegrino and C. Dovrolis, "Bandwidth requirement and state consistency in three multiplayer game architectures," in *in Proc. NetGames '03*, pp. 52–59, ACM Press, 2003.

[21] M. Roccetti, S. Ferretti, and C. Palazzi, "The brave new world of multiplayer online games: Synchronization issues with smart solutions," in *Object Oriented Real-Time Distributed Computing (ISORC), 2008 11th IEEE International Symposium on*, pp. 587 –592, may 2008.

[22] J. Smed, T. Kaukoranta, and H. Hakonen, "Aspects of networking in multiplayer computer games," 2001.