The DOOM III Network Architecture

March 6th 2006
J.M.P. van Waveren

Abstract

To immerse a player into a highly interactive game environment with fast-paced action, a computer game must deliver a consistent and responsive experience. However, this may be difficult to realize when a computer game is played over a network connection because network environments introduce various constraints such as an always limited amount of bandwidth that can be used to share information, and a delay before information arrives at its destination, also known as latency or lag. Any network architecture for a computer game implements a trade between: consistency, responsiveness, bandwidth and latency requirements. Finding the right balance depends on the type of computer game and the network environment. This paper presents the network architecture implemented for the first person shooter DOOM III. This architecture improves upon previous network architectures used in the computer games Quake, Quake II, and Quake III Arena.

1. Introduction

A First Person Shooter (FPS) is a game where the player moves through a real-time virtual environment, while looking at this environment from a first person perspective. The player looks through the eyes of the avatar controlled by the player. All the player can see of the avatar, are his hands and/or the weapon which he holds. In a first person shooter the most important tasks are staying alive and eliminating virtual opponents with a variety of weapons. First person shooters are highly interactive and typically fast-paced, emphasizing speed and accuracy. A good first person shooter is designed to immerse the player in the action.

Multiplayer gaming is about shared reality. Multiple players share experiences in a virtual environment, and although looking from different viewpoints, everyone sees the same changes and events take place within this virtual environment. Each player has his or her own computer and these computers are hooked up to each other through a local network or the Internet. The network is used to share information such that all players experience the same virtual environment.

To immerse a player into a highly interactive multiplayer game environment with fast-paced action, the game must deliver a consistent and responsive experience. It is very important that the players receive direct and immediate feedback to their actions. It is also important that all the players see events take place in the same way other players witness those events. However, network environments introduce constraints that make it difficult to realize a consistent and responsive game environment. First of all there is always a limited amount of bandwidth available and only so much information can be sent over the network. Furthermore, information sent over a network does not immediately arrive at the receiver. It takes time between one computer sending information and another computer receiving the information. This is also known as latency and in multiplayer games often referred to as lag [10]. Although the available bandwidth of Internet connections has significantly improved over the years, latencies have not improved as much. Even when fast glass fiber connections are used, a relatively large latency cannot be avoided if people located in different parts of the world want to play with each other.

In multiplayer games there will always be individuals that feel the need to overcome lack of skill with ingenuity, by exploiting bugs or holes in a network architecture, or by implementing various cheats. When designing a network architecture it is important to consider the cheats to which the system may be vulnerable. Another important issue is security [27, 28, 29]. Individuals may try to take down servers or otherwise disrupt games. Perhaps even worse, individuals may try to use the game as a tool to attack computers on a network or the Internet. A robust network architecture can withstand attacks and abuse. Last but not least portability may play a role in the design of a network architecture. Some network models are better suited for cross platform multiplayer gaming than others.

The game state in a first person shooter is typically modeled as a list of game objects or entities. Players, enemies, projectiles, doors etc. are all entities in the game. Instead of treating all of these differently, as special purpose elements, it is convenient to bind them together into a system that provides a common structure and common methods of communication. Class hierarchies and functional components are typically used to model the different entities [5]. Networking in first person shooters is all about synchronizing the state of multiple copies of the same game entities such that all players experience the same changes and events in the virtual environment. Some network models require all players to manage and maintain their own copy of all game entities where the same rules and methods are used to advance the state of these objects synchronously. Other network models continuously communicate changes to the state of entities over the network.

1.1 Previous Work

Peer-to-Peer

The networking engine in the original DOOM (1994) is a peer-to-peer system. Each player in the game is an independent "peer" running its own copy of the game. This peer-to-peer system can also be considered a synchronous networking system. All players run their own copy of the game synchronously. Periodically (every 1/35th of a second), the input from a player (from keyboard, mouse, etc.) is sampled and placed into a tick command. This is a simple structure which stores how the player wishes to move: there are fields for forward/backward movement, sideways

movement (strafing), turning, and actions such as "use" and "fire". The tick command is transmitted to all other players in the game. The networking engine periodically checks for newly received packets. Tick commands from other players are stored in a buffer. When tick commands from all players have been received, the game advances. The game advances independently from generating tick commands such that a player may generate commands to be used several ticks into the future, even though the game has not yet advanced to that point. As a result the game itself will not slow down, but there may be a delay between pressing a key, and the desired action occurring. The delay depends on the latency between players and the playability of the game is dependent on the player with the slowest connection.

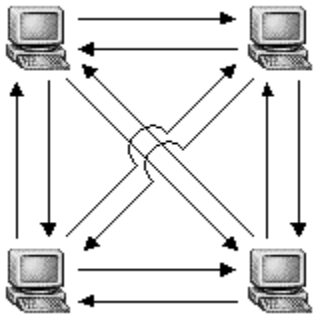

Fig. 1. Peer-to-Peer network with four players.

The advantage of synchronous networking is the simplicity of the system. The system is easy to implement and only simple small messages are sent around to other players. However, there are also several disadvantages.

All players have to maintain a perfectly synchronized copy of the game. If inconsistencies occur between the copies of the game the players will start to play their own "version" of the game which is no longer the same as the "version" of other players. In a complex game engine there are many places where inconsistencies can be introduced. Even a single bit difference in a floating-point number can escalate into huge changes in the game environment. To avoid inconsistencies the game state must be maintained without relying on any frame rate or computer hardware dependent input. For instance, the game state may be modified by introducing objects that move and shake based on the volume of sounds in the environment. Inconsistencies can easily be introduced if the sound volume in any way depends on a specific sound card or a different sampling rate based on the speed of the computer. Reading back data from the graphics card, for instance for hit detection, may also introduce inconsistencies because different players may use different graphics cards that produce different results. Inconsistencies can also be introduced by different drivers being used on various computers. A driver may for instance change the floating-point rounding mode of the floating-point unit (FPU) during game play which causes different results to be calculated on computers with different drivers. Uninitialized variables in the game code can also introduce inconsistencies. Such uninitialized variables should really be considered code bugs but during development they may occur frequently and as a result the networking may break as frequently.

In a complex game engine the cause for a specific inconsistency is typically hard to find because the time between the introduction of the inconsistency and the results being noticeable during

game play may be relatively long. If a single bit ends up being different between two copies of the game it may take some time before this difference escalates into a noticeable but significant difference to the players.

A synchronous network system is also not cross platform compatible. Players on different platforms will not be able to play with each other. On different hardware different assembler instructions may be used that produce (slightly) different results. Any such differences may introduce inconsistencies. Even if the underlying hardware is the same, for instance when using a combination of Microsoft Windows and Linux systems on x86 based hardware, inconsistencies may be introduced. Different compilers are typically used on different platforms and floating-point assembler instructions may be rearranged differently during optimization. Different floating-point instruction sequences usually produce results with different rounding, and such differences can also easily introduce inconsistencies.

Another problem with a synchronous network system is that the responsiveness and playability degrade quickly as the network latency of players increases. There is a full ping time (roundtrip latency) between sampling the player input and processing the input in the game. As a result the player only sees a change on screen a full ping time after issuing a command or pressing a key. On a Local Area Network (LAN) this is not a problem because ping times are usually below 10 milliseconds. However, on the Internet ping times can easily go up to 100 milliseconds or more. The responsiveness of the game is not good when there is 1/10th of a second between sampling input and visualizing a response on screen. On the Internet the ping times may also fluctuate or there may be congestion and sudden stalls that further degrade the playability because the game will stall as well.

In a game like DOOM III the game is updated at a fixed 60 Hz tick. Each player samples input at this rate and in a peer-to-peer system tick commands would be sent to all other players. Even though the tick commands result in rather small network messages, in a game with just four players the amount of traffic being generated already exceeds the bandwidth available on a 56k6 modem. Furthermore, the bandwidth requirements increase exponentially with the number of players in the game.

In a synchronous network system each player maintains a copy of the game and only tick commands are sent over the network. Therefore all players have to start the game together at the same time to be able to maintain a consistent copy of the game. New players cannot come and go as they please. This forces the development of additional functionality for match making or a lobby system [4], because players cannot simply search for an existing game and join at will.

Last but not least a synchronous network system creates opportunities for various cheats. A player cannot modify the game state directly because that would introduce inconsistencies. However, each player maintains a complete copy of the game and can use all kinds of visualization cheats to look at things the player is not supposed to see.

Packet Server

The peer-to-peer network model can be extended with a packet server. This model is basically the same as peer-to-peer model but with elements of a client-server architecture. Like the peer-to-peer model each player individually keeps track of the game state. However, each player no longer has a connection to all other players like in the original DOOM. In this model there is a 'packet server' running on the computer of one of the players. Everyone has one connection to this packet server. Tick commands are first sent to this server and the server relays all tick commands to other players. If one of the players has a high latency connection, only that player will experience a less responsive game, and the packet server will continue to send (possible duplicated) tick commands for that player to the other players. Using this model the players no longer need a low latency connection to all other players. The down-side of using a 'packet server' is that the game cannot continue if the player that runs the 'packet server' leaves the game. This network model does, however, solves some of the problems of the peer-to-peer network model but many of the other problems, as described above, remain.

Client-Server

The games Quake (1996), Quake II (1997) and Quake III Arena (1999) implement a so called client-server network architecture. In this model one computer is a "server" and is responsible for making all game play decisions. The other computers are "clients" and they are similar to dumb rendering terminals. A client sends input such as keystrokes and view angles to the server, and the client receives a list with entities to render. This network model does not suffer from de-synchronized network games, that could occur from clients disagreeing with each other, because the server is always the final authority. Players are also able to join and leave a game at any time because the players do not need to maintain the full game state. The server sends the list with entities to render as a sequence of game state updates also known as snapshots. Uncompressed, these updates can be quite large and require a lot of bandwidth. To reduce the bandwidth requirements the snapshots are delta compressed from the last acknowledged snapshot the client received. Furthermore, the updates are sent infrequently, typically at a rate between 10 and 20 Hz. However, for full interaction a higher frame rate is required at the client. To present a smoothly changing, interactive environment, the client interpolates between, or extrapolates from the last two snapshots.

Without special measures this network model would face the same problem as the peer-to-peer model described above when it comes down to the responsiveness and playability degrading as the network latency of players increases. The client sends input to the server where this input is processed, and the client only receives a response with the next snapshot from the server. In other words, using this network model there would also be at least a full ping time between sampling the input and seeing the results on screen at the client. To overcome this problem, client side prediction of the player movement is used. The input is not only sent to the server, but is also processed immediately at the client to move the view point and as such improve the perceived responsiveness. However, the rest of the environment visualized at the client is still the results of an interpolation between, or extrapolation from the last two snapshots. In other words, the player moves in the (predicted) present and what the player sees on screen is from the past.

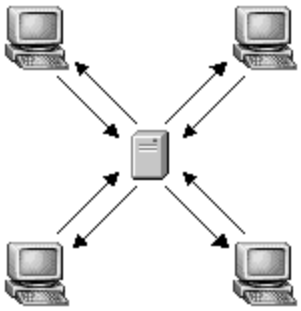

Fig. 2. Client-Server network with four players.

The client-server network model can also be considered an asynchronous network architecture. The upstream and downstream communication at the client and server is different. Furthermore, the player movement at the client is simulated in the present, while the game state maintained at the server is in the past, and the state of the environment visualized at the client is from even further back in the past. The network model used in the Quake series of games is very effective and has been used in many games since. This network architecture is much better suited for playing fast-paced first person shooters over the Internet than the peer-to-peer network model. There are, however, still several problems.

While the client-server network model allows players to join a game in progress, the game cannot automatically continue if the server runs on the computer of one of the payers and that player decides to leave the game. One solution is to use dedicated servers that never shut down (except for computer failures). However, if dedicated servers are not available it may be necessary to implement server migration to allow the game to continue even if the player that runs the server decides to leave the game.

With the client side prediction the player moves in the present while what the player sees on screen is from the past. As a result the player may be shooting at an opponent which is visualized at a position where the opponent was some time ago. However, the hit detection is performed at the server in the future. In other words the player has to predict where any targets will be some time in the future and has to lead such targets in order to hit them in the future. Even if a player has to predict only a 100 milliseconds ahead this can be tricky and requires some practice.

Some games, like Half-Life, try to alleviate this problem by implementing so called "lag compensation" [11]. The idea is to let a player shoot at opponents that are visualized at positions from the past and when the server does the hit detection it will go back in time to verify whether or not a target was hit in the past. For this purpose the server has to keep track of a history of past positions of all possible opponents. This technique does not only add considerable complexity, it also has several unwanted side effects. A player with a fast (low latency) connection will frequently experience inconsistencies. For example, the player may get shot after the player has moved safely around a corner. The worse the connection quality of the other players, the more often this will happen. In other words a player is no longer in control of the quality of his own game experience.

Lag compensation is also inherently flawed because it can cause events to happen that do not reflect the shared reality. For instance, take two players, one with a very low ping time and one with a high ping time. The players are circling around a tree, such that the tree always blocks the line of fire from the perspective of the player with the low ping. The player with a high ping moves in the present while looking at an opponent at a position from relatively far back in the past. From the perspective of this player the tree does not block the line of fire. With lag compensation the player with the high ping can now shoot and hit the player with the low ping because the server does hit detection with opponents positioned in the past while keeping the position of the player in the present. Obviously in this example the player with the low ping can be shot which from the perspective of that player should not be possible. In this particular example the problem can be alleviated by not only performing hit detection at the server using the position of an opponent in the past, but also checking for a clear line of sight with the opponent in the present. A hit is then only recorded when the opponent is hit in the past, and there is also a clear line of sight in the present. Although the additional line of sight test may make it less likely for players to notice inconsistencies in some situations, it by no means solves the problem in that inconsistencies may still occur in other situations.

Even if lag compensation is considered better than the alternative, it only addresses part of the problem. It is not practical for the server to roll back everything in time. So even if the positions of opponents are rolled back in time for hit detection, all other interaction at the client is between a player that moves in the present and an environment that is from the past, while the server verifies all other interaction in the future.

In Quake III Arena the game code (which runs at the server) and the visualization and prediction code (which runs at the client) are separate modules. These modules are often referred to as the "game code" and "client game code". Although these modules are separate, each must be fully aware of the implementation of the other in order to keep the representation of the game reasonably synchronized. Such a strong coupling between separate modules is undesirable because it makes extending the game difficult. This separation, yet strong coupling also makes it harder for developers that want to use the Quake III Arena engine to implement a single player experience. Several times developers have released two executables. One executable for multiplayer which is based on the original Quake III Arena source code with the separation between the modules. Another executable provides the single player experience and bypasses the separation which makes it easier to directly visualize things from the game code.

To present a smoothly changing environment the Quake III Arena client interpolates between, or extrapolates from the last two snapshots. Such extrapolation can produce unrealistic results unless the physics and game rules are taken into account. For instance, extrapolation of trajectories without collision detection can cause objects to move into or through other objects or walls. Proper extrapolation introduces additional complexity and more code at the client that needs to be in sync with the code that runs at the server.

In a complex game engine interpolation can be difficult because the state of an entity can be complex and proper interpolation is often expensive or ill-defined. For instance, a skeletal animation system may be used for animating characters, and arbitrary bone modifications may be applied to achieve specific effects. First of all the complete skeleton would need to be sent over

the network for proper interpolation. Furthermore expensive spherical linear interpolation between quaternions would be required to interpolate between the skeletons from the last two snapshots.

In Quake III Arena a snapshot for a client only contains the state of those entities that are in the Potentially Visible Set (PVS) [34] of the client. Furthermore, the entity states in a snapshot can only be delta compressed relative to the entity states from a previous snapshot. As a result the state of entities that continuously leave and enter the PVS cannot be delta compressed. Whenever an entity enters the PVS, it's state has to be sent in full with the next snapshot. In an environment with a lot of entities that need to be synchronized over the network this can cause a lot of bandwidth to be consumed.

Most network traffic in Quake III Arena is generated by sending unreliable messages. There is typically no point in resending the same snapshots if they are dropped because the information they contain becomes out of date very quickly. However, reliable messages can also be used in Quake III Arena for specific, usually infrequent, updates. The only way to send reliable messages in Quake III Arena is through the use of server commands. These server commands are string based and inefficient. However, it can be very convenient to synchronize a small part of the game state through reliable messages. Using string based server commands like in Quake III Arena introduces considerable overhead.

To write entity states to snapshots, Quake III Arena uses a fixed entity state structure with fixed data fields. The advantage is that all entities have the same structure and a single optimized routine can be used for the delta compression. On the other hand different types of entities may maintain rather different state variables and developers often ended up re-using entity state structure fields for different purposes on different entities. In a game with many different entities that maintain completely different state variables it quickly becomes impractical to use a fixed entity state structure for all entities.

The DOOM III network architecture presented here is similar to the network model of Quake III Arena but tries to address several of its problems.

1.2 Layout

Section 2 provides an overview of the DOOM III network architecture. Section 3 describes the data communicated between the client and server. Section 4 describes all forms of compression that are used to significantly reduce the amount of data that needs to be communicated over the network. Section 5 shows some of the implementation details. The results are presented in section 6 and several conclusions are drawn in section 7. Suggestions for future work are presented in section 8.

2. Network Architecture

Just like Quake III Arena, the network architecture presented here is based on the client-server model. The server writes snapshots of the game state and sends these to a client. A snapshot for a client contains the compressed state of all entities that are currently in the Potentially Visible Set (PVS) [34] of the client. Each client samples input (from keyboard, mouse etc.) and sends the player's intentions to the server. Figure 3 shows an overview of the client-server architecture.

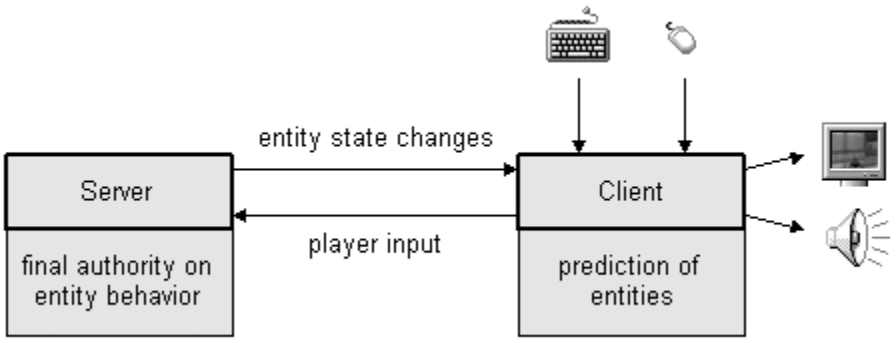

Fig. 3. Overview of client-server architecture.

Except for player input everything in a first person shooter is deterministic. Even random number generators are pseudo random and always produce the same sequence of "random" numbers based on an initial value. Furthermore the DOOM III game state is always updated at a fixed 60 frames per second independent from the speed of the computer. As such the engine always produces the same results based on the same player input. It makes sense to use this determinism to replicate information at the client. To present a smoothly changing environment the Quake III Arena client interpolates between, or extrapolates from the last two snapshots, and client side prediction of the player movement is used to improve the perceived responsiveness. The system presented here uses prediction on all entities in the PVS of the client to both improve the responsiveness and to present a smoothly changing environment. This includes prediction of the player movement because the avatar controlled by the player is also an entity in the PVS of the client.

The server progresses the game without waiting for input from players. The server duplicates old player input if no new input has arrived in time to process the next game frame. The client tries to make sure the server always has new input to advance the game state at the server. As such the time at the client is ahead of the server time. The client runs just far enough ahead such that input from the player can be processed immediately at the client and can be sent over the network to the server where it arrives before the server needs to process the input to advance the game. Figure 4 shows a timeline with the server time, the client time, and the time of snapshots that arrive at the client.

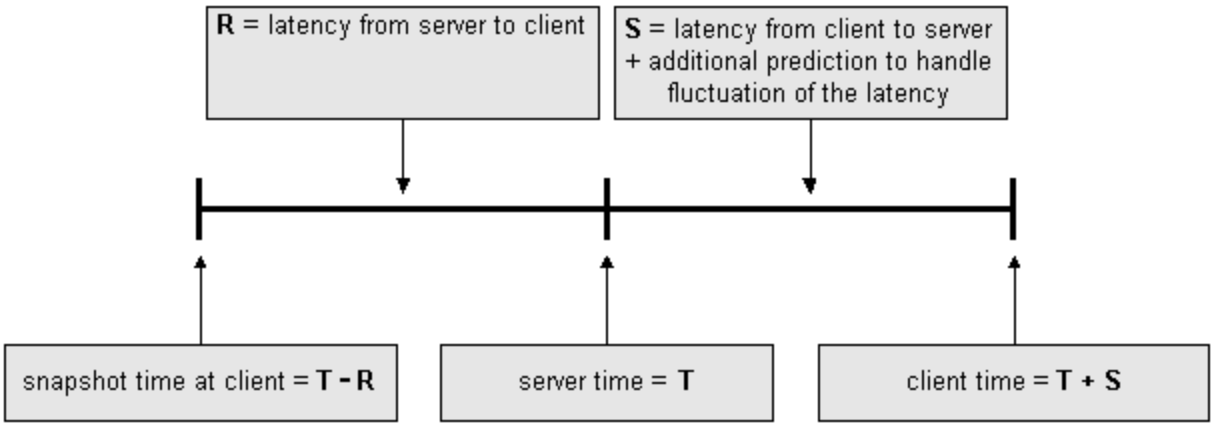

Fig. 4. Client-server timeline.

Ping times may fluctuate on the Internet. For this reason a client does not just run far enough ahead of the server such that input arrives in time at the server based on the current or average latency. The client runs a little further ahead such that input arrives in time even if there is some fluctuation in the latency of network messages sent to the server.

The client is able to run ahead of the server by using prediction to advance the state of entities. The server sends snapshots to the client at a rate between 10 and 20 Hz. While no new snapshot has arrived the client predicts the state of entities on a frame to frame basis. Upon receiving a snapshot the client overwrites it's entity states with the entity states from the snapshot. A snapshot the client receives is from at least a full ping time in the past relative to the current time at the client. As a result the client temporarily moves back in time when it processes the snapshot. The client then has to quickly re-predict ahead from this state up to the current client time, from where the client can continue the prediction on a frame to frame basis. Figure 5 shows the prediction at the client.

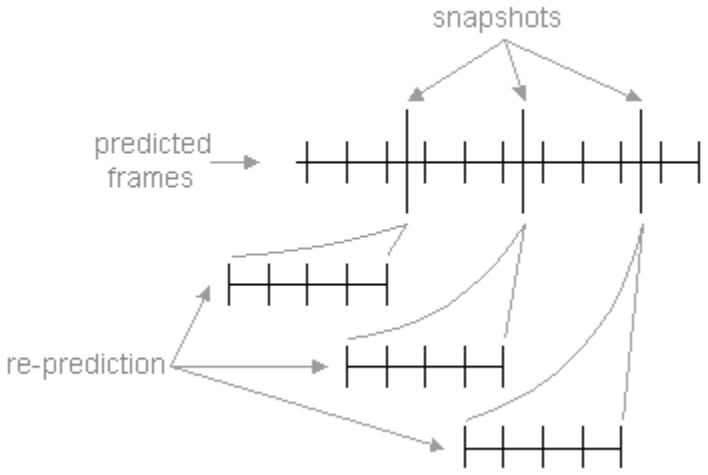

Fig. 5. Prediction at the client with a snapshot rate at 20Hz and a ping of around 80 milliseconds.

As stated above everything in a first person shooter is deterministic except for player input. Fortunately the player movement as a result of the input is predictable to a degree. The game advances at a fixed 60 frames per second and input is sampled at the same frequency. Players typically do not tap keys 60 times per second and do not change their movement direction every frame. As a result using the same input for several game frames is usually not too far off from what really happened. Furthermore the server sends the most recent input from other players with snapshots. This input from other players is used by a client to accurately predict other players. The input is processed by a client in the same way the server does, and the same player physics is used to move the predicted players through the virtual environment.

Unlike Quake III Arena there is no time difference between what the player sees on screen and the player movement through the environment. Therefore the player does not need to lead targets because the system does the prediction for the player. The server sends the most recent input from other players with snapshots and the same game code (including physics) is used at both the client and the server to move players through the environment. Therefore the system can typically do better prediction of other players than the player is able to do by leading targets. The Quake III Arena bots show that computer algorithms can be fairly good at predicting player movement. Even when using weapons that fire slow moving projectiles the bots at a higher skill level show exceptional accuracy when aiming. The bots in Quake III Arena use an approximation of the player physics with collision detection to predict where players will be some time in the future.

General prediction of all entities in the client PVS works well because the same game code and algorithms (including physics) are used at both the server and the client to advance the state of entities. This is also known as dead reckoning [9]. Unlike Quake III Arena, the game code, which runs at the server, and the client visualization and prediction code are not in separate modules. The same module is used both at the server to advance the state of entities, and is also used at the client to visualize and predict the state of entities. The server runs the game code just like in single player mode without modifications. The client runs the same game code to advance the state of entities for prediction. Using a single module allows development of a single player game or new features without being forced to make things work over a network.

3. Client-Server Communication

The User Datagram Protocol (UDP) [32] is used to send messages over a network or the Internet. Because of the highly interactive and fast-paced nature of first person shooter games there is no point in using a reliable protocol with guaranteed message delivery like the Transmission Control Protocol (TCP) [33]. The server continuously sends the states of entities over the network. Resending the same states when network messages are dropped is pointless because by the time such messages do arrive (after resending them), the entity states they contain are already out of date. With UDP, messages may arrive out-of-order with duplicates. Furthermore, messages may not arrive at all but the content of messages that do arrive is never corrupted. To deal with the characteristics of a network using UPD messages, the network architecture presented here implements several layers on top of UDP as shown in figure 6.

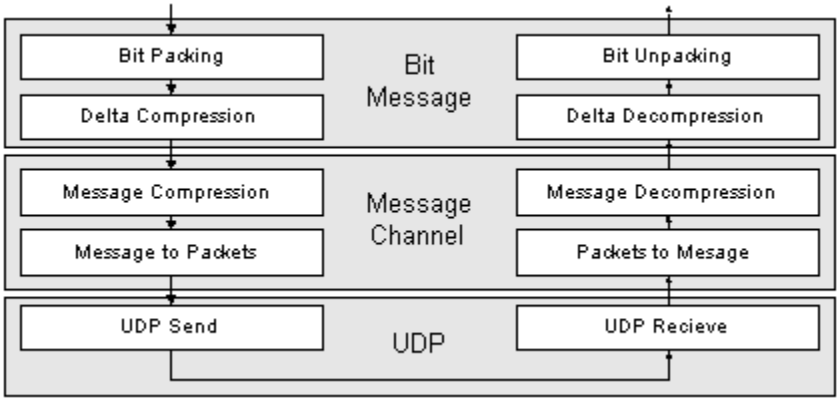

Fig. 6. Network layers.

The message channel implements one connection, either from a client to the server or the other way around, and provides functionality for sending both unreliable and reliable messages. These messages are guaranteed to arrive in-order, without duplicates and their content is never corrupted. However, the unreliable messages may be dropped while the reliable messages are guaranteed to always arrive. Most data like snapshots and player input is sent over the network with unreliable messages. Reliable messages are only used for certain acknowledgements and small critical updates that need to be processed before any unreliable updates. The message channel is designed to handle reliable messages that are very small compared to unreliable messages. The presented network system produces a continuous stream of unreliable messages. Snapshots are sent at 10 to 20 Hz from the server to the client, and player input is sent more frequently from the client to the server. Because there is a continuous stream of unreliable traffic, the reliable messages simply piggy back on unreliable messages. The message channel uses a brute force approach to send reliable messages. Reliable messages are buffered and each reliable message is sent with every unreliable message until the reliable message has been acknowledged by the receiver. As such the message channel guarantees that a reliable message arrives before the first next unreliable messages comes through.

This brute force approach to reliable messages works well. The client does not send reliable messages frequently. When a reliable message is sent, the message is usually very small so the

brute force approach is not an issue there. Furthermore, the client sends unreliable messages at a rate 3 to 4 times faster than the server sends unreliable messages. As such, there is no need for a time-out on a reliable message from the server because if a reliable is not acknowledged by any of the next couple of client messages, the message can be considered dropped and should be resent anyway.

The message channel is "rate controlled" and can notify higher layers when the network connection becomes satisfied and additional traffic would exceed the available bandwidth. The message channel also breaks up large messages into packets if they would otherwise cause fragmentation. Furthermore, the message channel keeps track of an unique identification for the computer at the other end of the connection. The Internet Protocol (IP) port number cannot be used as an unique identification because some routers may periodically change the IP port [6].

The message channel uses compression to reduce the size of all messages. Furthermore, the bit message layer implements two forms of compression that are required to significantly reduce the amount of data that needs to be communicated over the network. Before going into the details of compression, the next section describes the data being sent over the network.

3.1 Unreliable Message Headers

Snapshots and player input are sent over the network with unreliable messages. All unreliable messages from the server to the client and vice versa have a header. Figure 7 shows the header for unreliable messages from the server to the client. Figure 8 shows the header for unreliable messages from the client to the server.

32 bits	game id
8 bits	message type

Fig. 7. Unreliable messages header from server.

32 bits	sequence number of last recieved server message
32 bits	game id
32 bits	sequence number of last recieved snapshot
8 bits	message type

Fig. 8. Unreliable messages header from client.

The server uses an unique identification number for every game being played which usually happens on a per map basis. This identification number is used to make sure the clients use the right settings, and have the right map loaded for the current game. The server sends this game identification number with every unreliable message. The server also sends the type of the unreliable message.

The client also sends a header with every unreliable message. This header stores the sequence number of the last received server messages and the game identification number. The server uses the game identification number sent by the client to check whether or not the client is in the right game. If the client is not in the right game the server sends an unreliable message to the client with instructions on which map to load and additional settings. Unreliable messages are not acknowledged, but the client sends the sequence number of the last received server messages to

allow the server to verify whether the client received the unreliable message with instructions to load the right game. The client also sends an acknowledgement for the last received snapshot with every unreliable message. This acknowledgement is used for the delta compression described in section 4. Just like for unreliable messages from the server the client sends the type of the unreliable messages.

3.2 Snapshots

The most important traffic from the server to the client is in the form of snapshots. Such snapshots contain a collection of data but most importantly they communicate the states of entities to a client. Figure 9 shows a snapshot and figure 10 shows how a snapshot goes through the network layers.

32 bits	game id
8 bits	message type: snapshot
32 bits	snapshot sequence number
32 bits	game frame number
32 bits	game frame time
8 bits	number of duplicated user commands
16 bits	client ahead time
variable # bits	delta compressed entity states
variable # bits	delta compressed PVS bit string
variable # bits	delta compressed game & player state
variable # bits	latest user commands from the other clients in the PVS

Fig. 9. Snapshot Message.

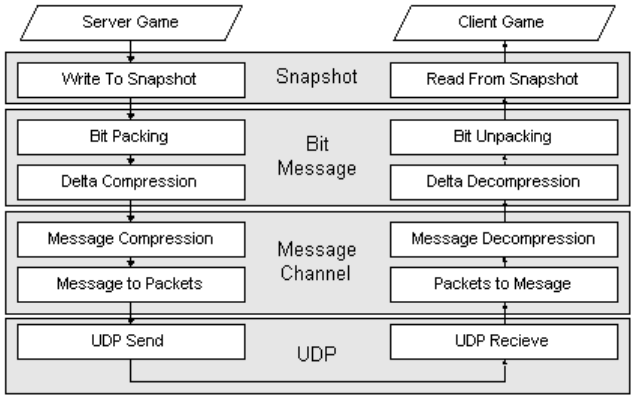

Fig. 10. Server to client snapshot pipeline.

Every snapshot message has a sequence number which allows the client to acknowledge the snapshot for delta compression as described in section 4. Next the snapshot message stores the game frame number and the game time when the snapshot was made. The snapshot also tells the client how many user commands were duplicated since the last snapshot. If the client does not predict far enough ahead and the client's user commands do not arrive in time the server will duplicate previous user commands. The client should avoid this. The number of duplicated user commands is not used directly but can be displayed during development. The server also tells the client how far ahead of time the user commands from the client arrive. This time should ideally be close to zero, but always positive, and the client can adjust the prediction time accordingly.

The most important data in a snapshot is a set of delta compressed states of entities that are in the PVS of the client. Unlike Quake III Arena there is no fixed entity state where structure fields may need to be reused for different purposes on different entities. The network system presented here uses a bit message to store an entity state. Entities can write any variables that need to be synchronized to this state. It is even possible to change the structure of the state during game play but this does decrease the performance of the delta compression.

Entities that did not change are not stored in the snapshot. However, the client still needs to know which entities are considered in the PVS by the server. For this reason a bit string is stored which tells which entities are considered in the PVS.

Aside from the entity states there is also one delta compressed state which is not tied to the PVS and always stored in the snapshot. This state is used to communicate general game and player state information. Furthermore a snapshot stores the most recent user commands from other players that are in the PVS of the client. These user commands are used by the client to improve the prediction of other players.

3.3 User Commands

The most important traffic from the client to the server is in the form of user commands. Such user commands contain the intentions of the player. Figure 11 shows an user command message and figure 12 shows how a user command message goes through the network layers.

32 bits	sequence number of last recieved server message
32 bits	game id
32 bits	sequence number of last recieved snapshot
8 bits	message type: user command
16 bits	client prediction in milliseconds
32 bits	game frame number
8 bits	number of user commands
variable # bits	delta compressed user commands

Fig. 11. User Command Message.

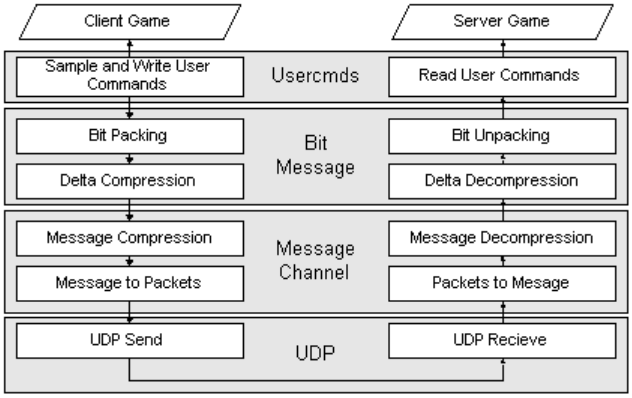

Fig. 12. Client to server user command pipeline.

With a user command message the client sends a time which represents how far the client predicts ahead relative to the server. This time is not directly used by the server but can be displayed at the server during development. A user command message may contain multiple user commands. The message stores the game frame number of the first user command and the number of user commands. The first user command is for the given frame number, the next for the following frame etc. The message then stores the actual user commands. These user commands are delta compressed in sequence. The second user command is delta compressed relative to the first, the third relative to the second etc.

4. Compression

Snapshots only contain states of entities in the PVS of a client and snapshots are sent at a frequency between 10 an 20 Hz. The snapshots still consume a lot of bandwidth without any form of compression. Several forms of compression are required to significantly reduce the amount of traffic.

4.1 Bit Packing

Bit packing is a form of compression where superfluous bits are removed from variables before sending them over the network [3]. For instance, the health of a player may be stored in memory as a 32-bit integer for computational efficiency. However, the health usually ranges from 0 to 100 and only 7 bits are required to store a health value. There is no need to send the remaining 25 bits over the network. By sending a health value over the network with just 7 bits no information is lost while the amount traffic is reduced with more than 75%.

Variables can also be quantized to reduce the amount of traffic that is generated. For instance, floating point values can be transmitted in a format which uses less bits by specifying the number of bits used for the mantissa and exponent. In this case information is lost because the floating point values are communicated in a format with less precision. However, for many variables the reduced precision is not a problem. The positions of objects are sent over the network in full precision because otherwise collisions may be missed during prediction. However, the velocities of objects can usually be transmitted as 16-bit floats.

Angles in degrees are good candidates for quantization. The value of a floating point angle ranges from 0 to 360 and the value is best represented with the same precision across the whole range. As such an angle can be scaled to the range 0 to 65535 and reduced to a 16 bit integer. When the value is unpacked it is scaled back to a floating point number in the range 0 to 360.

Sometimes it is also convenient to send an approximate direction over the network. For instance, the initial direction for a particle effect does need to be communicated with full precision. A normalized direction vector can usually be approximated with very few bits. For instance, just 3 bits for each coordinate axis.

4.2 Delta Compression

In a game like DOOM III there are many entities with many state variables in any given scene. Bit packing alone is not enough to bring down the size of snapshots. All state variables can change but typically only a subset of the variables changes over time. As such it makes sense to only write state variable changes to snapshots. For this reason the network system presented here uses delta compression. Bit packing is used before delta compression because only the changes to the quantized variables need to be transmitted.

Two kinds of delta compression are supported. First of all variables can be delta compressed relative to an initial or common value. If a variable has a specific value for most of its life time the server and client can assume this value until it changes. Variables can also be delta

compressed relative to a previous value which is known to be replicated at the receiver. If the sender knows the current value of a variable is already replicated at the receiver, the value does not have to be transmitted again. The delta compression uses one bit to tell whether or not a variable changed. If the value of a variable changed a single bit, set to 1, is written followed by the bit packed variable value. If the value of a variable did not change only a single bit, set to 0, is written.

In Quake III Arena entities can only be delta compressed relative to a previous snapshot. The delta compression presented here works relative to a common base with entity states maintained by both the server and the client. This common base contains states for all entities in the game. Entities are delta compressed relative to this common state which allows proper delta compression even when entities continuously leave and enter the PVS. After creating and sending a snapshot the server buffers the snapshot. The client also keeps track of the recently received snapshots. With each user command message the clients sends an acknowledgement for the last received snapshot. User commands are sent at a higher frequency than snapshots so even with packet loss most snapshots are acknowledged. When the server receives an acknowledgement for a snapshot it applies the snapshot to its common state for that client. The server then also sends a reliable message to the client telling the client to apply the same snapshot to it's common state. Reliable messages are guaranteed to arrive before any new unreliable data, so the acknowledged snapshot will be applied at the client before any new snapshots are processed. This way both the server and client maintain a synchronized common base with entity states that can be used for delta compression.

Entities that have not change from the common state are not written to a snapshot at all to save even more bandwidth. However, a client still needs to know which entities are considered in the PVS by the server because the client should only render those entities on screen. Entities are typically numbered and the server could write the numbers of all entities that are in the PVS to the snapshot. In a game like DOOM III there can be many entities in the PVS in any given scene and sending all those numbers would significantly increase the size of snapshots. Instead of sending the numbers of the entities, a bit string can be written to a snapshot. This bit string has a bit for each entity where bit offset denotes the entity number. A bit is set to 1 if the entity is in the PVS of the client and a bit is set to 0 if it is not. With a maximum of 4096 entities in the game this requires 512 bytes. However, as a player moves through the environment typically only a few entities enter and/or leave the PVS. As such this bit string can be delta compressed in fixed chunks of for instance 32 bits. A 32 bit chunk is then only sent over the network if one of the entities from that chunk entered or left the PVS. If no entities entered or left the PVS this results in just 16 bytes being written to the snapshot.

4.3 Message Compression

The message channel compresses all reliable and unreliable messages. The delta compressed bit packed data is not a stream of fixed length words. The delta compression introduces single bits and the bit packed variables can use an arbitrary number of bits. As a result regular entropy encoding is difficult because entropy encoders typically assume a word length of a fixed number of bits.

The delta compression writes out a single zero bit for any variable that did not change. Because only few variables change over time the delta compression introduces a lot of sequences of zero bits. For this reason 3-bit word length zero based run-length compression is used. The compressor reads 3 bits at a time and if the bits are not all zero the same bits are written out without changes. If the 3 bits are all zero the compressor keeps reading words of 3 bits at a time until they are no longer all zero. The compressor then writes out 3 zero bits followed by 3 bits that represent the number of times 3 zero bits were read in succession. Using a 3-bit word length results in a maximum compression ratio of 4**:**1. A different number of bits than 3 could be used for the word length but for the DOOM III entities using 3 bits results in the best compression ratios in practice.

The compression ratios for snapshots can generally be improved by writing entity class variables to a snapshot in order of the frequency at which they change. This way the variables that change infrequently are grouped together which results in long sequences of zero bits.

5. Implementation

Each entity class implements a WriteToSnapshot and ReadFromSnapshot member function. These member functions write and read entity class variables that need to be synchronized from the server to the client.

Write and read methods are used because this allows specific conversions to be implemented on a per entity basis. For instance, a rotation matrix can first be converted to a quaternion before writing the orientation to the snapshot. When the snapshot is read by the client the quaternion can be converted back to a rotation matrix. Certain state variables may also be derived from other state variables. Such derived state variables do not need to be synchronized and can be derived in the ReadFromSnapshot method.

Instead of read and write methods, tables could be used with entity class variables that need to be synchronized. Conversions and derived variables would need to be specified in these tables. A table would then be used by generalized routines to write and read the state variables. This would force these routines to know about conversions or derived variables that may be very specific to just one type of entity. Such a coupling is typically undesirable. The system presented here uses write and read methods for maximum localized flexibility.

```
class idMyEntity : public idEntity {
public:

    virtual void    WriteToSnapshot( idBitMsgDelta &msg ) const;
    virtual void    ReadFromSnapshot( idBitMsgDelta &msg );
    virtual bool    ClientReceiveEvent( int event, int time, const idBitMsg &msg );

    enum {
        EVENT_FIRST = idEntity::EVENT_MAXEVENTS,
        EVENT SECOND
    };

private:
    int             health;
}
```

```
void idMyEntity::WriteToSnapshot( idBitMsgDelta &msg ) const {
    msg.WriteBits( health, 7 );
}

void idMyEntity::ReadFromSnapshot( idBitMsgDelta &msg ) {
    msg.ReadBits( health, 7 );
}
```

The idBitMsgDelta class is initialized with an entity state from the common base. The class writes or reads a bit packed and delta compressed message. The class also writes out a new entity state which can be applied to the common state once a snapshot is acknowledged by both the client and the server.

The server can also send reliable events to update entities or to initiate specific effects. The server can send an event to clients for a specific entity by calling a ServerSendEvent method on the entity. These entity events are not affected by the PVS of a client and will always be sent over the network.

```
void idEntity::ServerSendEvent( int event, const idBitMsg *msg, bool saveEvent ) const;
```

The 'event' parameter specifies the type of event. Each entity can handle an arbitrary number of different events. The event parameters are written to an idBitMsg which allows the parameters to be bit packed to save bandwidth.

If the 'saveEvent' option is set then the event is saved for clients that connect late. Saved events are buffered and also sent to any client that connects late so all clients always receive the events no matter what time they join the game. Such saved events can be used for entities that exist throughout the whole game and do not change continuously but go through a well defined sequence of transitions. Whenever a transition takes place an event is sent to the client to execute the particular transition.

To handle such events each entity class has a method which is called whenever an event arrives at the client for an object of that entity class.

```
bool idMyEntity::ClientReceiveEvent( int event, int time, const idBitMsg &msg ) {
    switch( event ) {
        case EVENT_FIRST: {
            // read parameters from 'msg' and handle event
            return true;
        }
        case EVENT_SECOND: {
            // read parameters from 'msg' and handle event
            return true;
        }
        default: {
            return idEntity::ClientReceiveEvent( event, time, msg );
        }
    }
    return false;
}
```

Even though the events are reliable the entity can still choose to ignore the event if it is from too far back in the past. In DOOM III there is no functionality implemented to send unreliable

events. However, implementing such events can easily be done in the game code by writing the events just once to snapshots. Such unreliable events can also be bound to the PVS of the client if desired.

In Quake III Arena implementing and tweaking an entity that works over the network sometimes requires changing almost a dozen source code files. With the network architecture presented here there is no separation in modules. As such the implementation of everything necessary to simulate and visualize an entity is localized and typically involves no more than two source code files.

6. Results and Discussion

In DOOM III the bit packing typically reduces the amount of traffic with 10 up to 15%. The reason for this relatively small reduction is that a lot of positions and orientations of entities are synchronized as full 32-bit floating point variables without any reduction.

In theory the maximum compression ratio of plain delta compression is 32:1 because the maximum size of variables used in DOOM III is 32 bits and the delta compression writes out a single zero bit for each variable that did not change since a previous update. In practice the delta compression does much better and reduces the traffic with 90 up to 100%. The reason for this large reduction is that entities in the environment that do not change at all are omitted completely from the network stream and no zero bits are transmitted. If entities that do not change at all would still be synchronized the delta compression would turn updates for such entities into sequences of only zero bits and the reduction in network traffic would only be 80 to 90%.

The 3-bit word length zero-based run-length compression has a theoretical maximum compression ratio of 4:1. In practice the run-length compression reduces the network traffic with 15% up to 50%.

Unlike with Quake III Arena, developers are not forced to make all entities network capable. The presented network architecture allows a single player experience to be developed independently, without having to worry about networking. This is generally a big advantage. However, entities developed for a single player experience may not be suitable or optimized for networking. This may result in a fair amount of work when a developer tries to build a multiplayer experience upon a single player game.

The bit packing is used to reduce the amount of entity state information being written to snapshots. The number of bits used to synchronize integer variables can be decreased and even the number of bits used for the mantissa and exponent of floating point values can be specified. However, reducing the entity states on a bit level is not trivial and requires thorough knowledge of the information being synchronized. For instance, reducing the number of bits used to synchronize physics positions is usually a bad idea because predicted collisions may be missed if positions are not synchronized with enough precision.

The DOOM III engine as a whole is more flexible than the Quake III Arena engine. There are more different entities that can operate and change in different ways. To cope with this increased

flexibility the DOOM III network system is also more flexible than the Quake III Arena networking. The additional flexibility comes with more choices to be made and as such it can take some time to get familiar with all the different options. Synchronizing class variables on a per entity basis also requires more programming than writing code to synchronize a single fixed entity state. However, the additional flexibility is required because entities are versatile and may have many different state variables that do not all fit well in a fixed state structure.

While no new snapshot has arrived, the prediction at the client is cheaper than advancing the game at the server because only entities in the PVS of the client are predicted. However, whenever a new snapshot arrives the states of entities in the PVS of a client are overwritten with the states from the snapshot. The client then has to re-predict up to the current client time because the snapshot contains entity states from the past. The snapshots are from at least a full ping time in the past, and the client may need to re-predict quite a few game frames in one go. Even though only entities in the PVS of the client are predicted, this prediction can be expensive at times. Furthermore, the prediction becomes more expensive as the latency increases. Generally the PVS is limited enough such that the prediction does not cause any slowdowns. However, the multi-frame prediction happens whenever a new snapshot arrives, and snapshots typically arrive at a lower frequency than the client renders images on screen. As a result more processing power is required every few rendered frames which may degrade the display of smooth motion through the virtual environment.

7. Conclusion

Any network architecture implements a trade between: consistency, responsiveness, bandwidth and latency requirements. Finding the right balance depends on the type of game and the network environment.

The network architecture presented here overcomes several of the problems of the network architecture used in Quake III Arena. There is no server and client game separation which makes development easier, and a single player game can be implemented without having to worry about networking. There is no time difference between the player movement through the environment and what the player sees on screen. Entities are not synchronized through a fixed entity state structure but instead arbitrary entity state variables can be synchronized over the network. The network system is also generally more efficient than the Quake III network system.

Even though the efficiency of the network architecture improved, multiplayer games in DOOM III usually generate more traffic than in Quake III Arena. In DOOM III there are simply many more entities with many state variables in any given scene and synchronizing all those variables requires more bandwidth. For instance, while in Quake III Arena the lighting was baked into the levels, the lighting in DOOM III is fully dynamic and every single light in the environment can be changed at run-time and may therefore have to be synchronized over the network.

8. Future Work

Snapshots are delta compressed relative to a common base with entity states maintained by both the server and the client. When a map is first loaded the common base is empty. As a result the full state of any entity that enters the PVS of a client for the first time is sent over the network. However, a lot of entities are loaded from a map file and they either do not change at all, or only part of their state changes during game play. The common base with entity states could be initialized with the states of entities from right after a map is loaded. These states are always the same and as such the common base at the client and server stay synchronized when initialized with these states.

In DOOM III reliable messages can be used to transmit events and small, but significant updates of entities. However, some small updates could also be sent over the network with unreliable messages instead. Such unreliable messages could be used for events and updates that are only noticeable over a short period of time and do not disrupt game play if they never arrive at a client. For instance, bullet impact effects on walls could be communicated with unreliable messages. These effects disappear quickly and if some of these effects never arrive at a client it does not disrupt game play. Support for unreliable messages is easy to implement. These messages can be buffered in the game code at the server. All such buffered unreliable messages are then written to the first next snapshot and the buffer is cleared. If the snapshot message is dropped, the unreliable messages will never arrive at the client. However, packet loss is minimal on today's networks and the Internet, so most snapshot and unreliable messages will arrive.

All entities in the game are by default synchronized over the network when they enter the PVS of a client. For certain special effects it may be convenient to support entities that only live at the client. The server does not know about such client side entities and these entities are never synchronized over the network. The client can use these entities to display additional changes and events in the environment. Such client side entities can be derived from other entities that are synchronized over the network. These entities can also be created from additional reliable or unreliable messages sent from the server.

Not only the delta compressed states of entities that are in the client PVS are written to snapshots. There is also one delta compressed state synchronized over the network which is not tied to visibility. This state is used to communicate general game information and player state information over the network. Instead of using a single state for this purpose, it may be convenient to use multiple states for different purposes that are not tied to visibility. Using multiple such states allows individual states to be omitted if they are not relevant at certain points in time or if they are completely delta compressed away.

The client side prediction is implemented in the game code. As such developers can easily disable prediction on certain entities or specific aspects of some entities. For instance, the prediction of the position of opponents could be disabled and instead interpolation could be used. This allows a "lag compensation" approach [11] to be implemented without requiring changes to the network architecture. It is also possible to stop updating certain entities over the network at some point where the client side prediction completely takes over the rest of the simulation. For instance, the server could stop updating an enemy that dies and turns into a ragdoll. These

ragdolls are mostly used to present a realistic death animation and as such have no consequences for the actual game play. No longer updating such entities can save a significant amount of bandwidth.

Instead of a single snapshot stream the server can also send multiple snapshots to clients at different frequencies. This allows different entities to be updated at different frequencies. For instance, entities that are in the PVS but are very far away could be updated less frequently because state changes of such entities are less noticeable to a player. Sending updates less frequently obviously saves bandwidth. Certain entities can also be written to every other or every third etc. snapshot to save bandwidth.

When a new snapshot arrives the states of entities at a client are overwritten with the states from the snapshot and the client has to quickly re-predict up to the current time at the client. In DOOM III this re-prediction is cheap enough to not cause any problems. However, if this re-prediction turns out to be expensive at times the entity states can be advanced by taking larger steps through time. Instead of advancing the states one frame at a time at 60Hz, multiple frames could be merged to step through time more quickly. Obviously this will affect the accuracy of the prediction. However, taking larger steps through time can be done on a per entity basis and, for instance, only for entities where any prediction errors are not noticeable.

9. References

1. Networking and Online Games: Understanding and Engineering Multiplayer Internet Games
   Grenville Armitage, Mark Claypool, Philip Branch
   John Wiley & Sons, April 2006
   ISBN: 0-470-01857-7
   Available Online: http://eu.wiley.com/WileyCDA/WileyTitle/productCd-0470018577,descCd-description.html
2. Real-Time Strategy Network Protocol
   Jan Svarovsky
   Game Programming Gems 3, 2002
   Available Online: http://www.GameProgrammingGems.com
3. Bit Packing: A Network Compression Technique
   Pete Isensee
   Game Programming Gems 4, 2004
   Available Online: http://www.GameProgrammingGems.com
4. General Lobby Design and Development
   Shekhar Dhupelia
   Game Programming Gems 4, 2004
   Available Online: http://www.GameProgrammingGems.com
5. Component Based Object Management
   Bjarne Rene
   Game Programming Gems 5, 2005
   Available Online: http://www.GameProgrammingGems.com
6. Overcoming Network Address Translation in Peer-to-Peer Communications
   Jon Watte
   Game Programming Gems 5, 2005
   Available Online: http://www.GameProgrammingGems.com
7. A Reliable Messaging Protocol
   Martin Brownlow
   Game Programming Gems 5, 2005
   Available Online: http://www.GameProgrammingGems.com
8. Event Ordering and Congestion Control for Distributed Multiplayer Games
   Chris GauthierDickey, Daniel Zappala, Virginia Lo
   May 14, 2005
   Available Online: http://citeseer.ist.psu.edu/724490.html
9. Dead Reckoning: Latency Hiding for Networked Games
   Jesse Aronson
   Gamasutra, September 19, 1997
   Available Online: http://www.gamasutra.com/features/19970919/aronson_01.htm
10. A Look at Latency in Networked Games
    Jonathan Blow
    Game Developer Magazine, July 1998
    Available Online: http://www.flipcode.com/cgi-bin/fckb.cgi?showLink=246

11. Latency Compensating Methods in Client/Server In-game Protocol Design and Optimization
Yahn W. Bernier
Game Developers Conference, 2001
Available Online: http://www.gamasutra.com/features/gdcarchive/2001/bernier.doc
12. Prediction techiques in computer games
Joakim Bech
Master Thesis, Supervisor: Lennart Ohlsson, May 2005
Available Online: http://graphics.cs.lth.se/theses/projects/pticg/
13. The TRIBES Engine Networking Model
Mark Frohnmayer, Tim Gift
Game Developers Conference, pp. 191-207, 2000
Available Online: http://www.gamasutra.com/features/gdcarchive/2000/frohnmayer.doc
14. The Quake 2 Networking Data Flow
Q2DP
March 1998
Available Online:
http://www.gamers.org/dEngine/quake2/Q2DP/Q2DP_Network/Q2DP_Network.html
15. The Quake3 Networking Model
Brian Hook
Book of Hook, April 2004
Available Online:
http://www.bookofhook.com/Article/GameDevelopment/TheQuake3NetworkingModel.html
16. Unreal Networking Architecture
Tim Sweeney
Epic MegaGames, Inc., July 1999
Available Online: http://unreal.epicgames.com/Network.htm
17. A Distributed Multiplayer Game Server System
Eric Cronin, Burton Filstrup, Anthony R. Kurc
Electrical Engineering and Computer Science Department University of Michigan
UM EECS589 Course Project Report, May 2001
Available Online: http://warriors.eecs.umich.edu/games/papers/quakefinal.pdf
18. An Efficient Synchronization Mechanism for Mirrored Game Architectures (Extended Version)
Eric Cronin, Burton Filstrup, Anthony R. Kurc, Sugih Jamin
Electrical Engineering and Computer Science Department University of Michigan
In Kluwer MTAP, 2003
In Proc. NetGames2002, April 2002
Available Online: http://warriors.eecs.umich.edu/games/papers/mtap-tss.pdf
19. A Distributed Architecture for Interactive Multiplayer Games
Ashwin R. Bharambe, Je Pang, Srinivasan Seshan
School of Computer Science Carnegie Mellon University Pittsburgh, January 2005
Available Online: http://www.cs.cmu.edu/~ashu/papers/cmu-cs-05-112.pdf

20. Aspects of Networking in Multiplayer Computer Games
Jouni Smed, Timo Kaukoranta, Harri Hakonen
Proceedings of International Conference on Application and Development of Computer Games in the 21st Century, pp. 74-81, Hong Kong SAR, China, November 2001
Available Online: http://staff.cs.utu.fi/~jounsmed/papers/AspectsOfMCGs.pdf

21. A Review on Networking and Multiplayer Computer Games
Jouni Smed, Timo Kaukoranta, Harri Hakonen
University of Turku, April 2002
Available Online: http://staff.cs.utu.fi/~jounsmed/papers/TR454.pdf

22. An Experimental Estimation of Latency Sensitivity In Multiplayer Quake 3
G.J. Armitage
CAIA Technical Report 030405A, April 2003
Available Online: http://caia.swin.edu.au/reports/030405A/CAIA-TR-030405A.pdf

23. Dissecting Server-Discovery Traffict Patterns Generated By Multiplayer First Person Shooter Games
Sebastian Zander, David Kennedy, Grenville Armitage
NetGames, October 2005
Available Online: http://www.research.ibm.com/netgames2005/papers/zander.pdf

24. Towards a General Model of First Person Shooter Game Traffic
Philip A. Branch, Grenville J. Armitage
Swinburne University of Technology, Melbourne, Australia, December 2005
Available Online: http://caia.swin.edu.au/reports/050928A/CAIA-TR-050928A.pdf

25. Quake 3 Packet and Traffic Characteristics
M. D. Pozzobon
CAIA Technical report 021220A, December 2002
Available Online: http://caia.swin.edu.au/genius/021220A/

26. Enemy Territory Traffic Analysis
J. Bussiere, S. Zander
CAIA Technical Report 060203A, February 2006
Available Online: http://caia.swin.edu.au/reports/060203A/CAIA-TR-060203A.pdf

27. Designing Secure, Flexible, and High Performance Game Network Architectures
K. Jenkins
Gamasutra, December 2004
Available Online: http://www.gamasutra.com/features/20041206/jenkins_01.shtml

28. Security Issues in Online Games
Jianxin Jeff Yan, Hyun-Jin Choi
The Electronic Library, Vol. 20, No.2, 2002
International Conference on Application and Development of Computer Games in the 21st Century, Hong Kong, November 2001
Available Online: http://homepages.cs.ncl.ac.uk/jeff.yan/TEL.pdf

29. Security Design in Online Games
Jianxin Jeff Yan
The Chinese University of Hong Kong
Proceedings of the 19th Annual Computer Security Applications Conference, IEEE Computer Society, December, 2003
Available Online: http://homepages.cs.ncl.ac.uk/jeff.yan/acsac03.pdf

30. Cheat-Proofing Dead Reckoned Multiplayer Games
Eric Cronin Burton Filstrup Sugih Jamin
Electrical Engineering and Computer Science Department University of Michigan
In Proc. ADCOG 2003, January 2003
Available Online: http://warriors.eecs.umich.edu/games/papers/adcog03-cheat.pdf

31. Internet Protocol, RFC 760
J. Postel
USC/Information Sciences Institute, January 1980
Available Online: http://www.rfc-editor.org/rfc/rfc760.txt

32. User Datagram Protocol, RFC 768
J. Postel
USC/Information Sciences Institute, January 1980
Available Online: http://www.rfc-editor.org/rfc/rfc768.txt

33. Transmission Control Protocol, RFC 761
J. Postel
USC/Information Sciences Institute, January 1980
Available Online: http://www.rfc-editor.org/rfc/rfc761.txt

34. A Survey of Visibility Methods for Networked Games
Matthew M. Trentacoste
Department of Computer Science University of British Columbia, 2002
Available Online: http://www.cs.ubc.ca/~mmt/538-final-paper.pdf