Latency Compensating Methods in Client/Server In-game Protocol Design and Optimization

Yahn W. Bernier (yahn@valvesoftware.com (mailto:yahn@valvesoftware.com)), 2001
Software Development Engineer

Contents

Overview

Designing first-person action games for Internet play is a challenging process. Having robust on-line gameplay in your action title, however, is becoming essential to the success and longevity of the title. In addition, the PC space is well known for requiring developers to support a wide variety of customer setups. Often, customers are running on less than state-of-the-art hardware. The same holds true for their network connections.

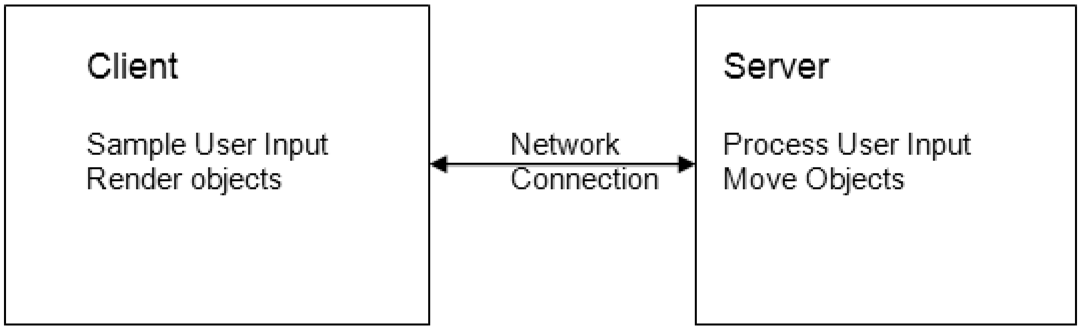
While broadband has been held out as a panacea for all of the current woes of on-line gaming, broadband is not a simple solution allowing developers to ignore the implications of latency and other network factors in game designs. It will be some time before broadband truly becomes adopted in the United States, and much longer before it can be assumed to exist for your clients in the rest of the world. In addition, there are a lot of poor broadband solutions, where users may occasionally have high bandwidth, but more often than not also have significant latency and packet loss in their connections.

Your game must behave well in this world. This discussion will give you a sense of some of the tradeoffs required to deliver a cutting-edge action experience on the Internet. The discussion will provide some background on how client / server architectures work in many on-line action games. In addition, the discussion will show how predictive modeling can be used to mask the effects of latency. Finally, the discussion will describe a specific mechanism, lag compensation, for allowing the game to compensate for connection quality.

Basic Architecture of a Client / Server Game

Most action games played on the net today are modified client / server games. Games such as Half-Life, including its mods such as Counter-Strike and Team Fortress Classic, operate on such a system, as do games based on the Quake3 engine and the Unreal Tournament engine. In these games, there is a single, authoritative server that is responsible for running the main game logic. To this are connected one or more "dumb" clients. These clients, initially, were nothing more than a way for the user input to be sampled and forwarded to the server for execution. The server would execute the input commands, move around other objects, and then send back to the client a list of objects to render. Of course, the real world system has more components to it, but the simplified breakdown is useful for thinking about prediction and lag compensation.

With this in mind, the typical client / server game engine architecture generally looks like this:



For this discussion, all of the messaging and coordination needed to start up the connection between client and server is omitted. The client's frame loop looks something like the following:

1. Sample clock to find start time
2. Sample user input (mouse, keyboard, joystick)
3. Package up and send movement command using simulation time
4. Read any packets from the server from the network system
5. Use packets to determine visible objects and their state
6. Render Scene
7. Sample clock to find end time
8. End time minus start time is the simulation time for the next frame

Each time the client makes a full pass through this loop, the "frametime" is used for determining how much simulation is needed on the next frame. If your framerate is totally constant then frametime will be a correct measure. Otherwise, the frametimes will be incorrect, but there isn't really a solution to this (unless you could deterministically figure out exactly how long it was going to take to run the next frame loop iteration before running it...).

The server has a somewhat similar loop:

1. Sample clock to find start time
2. Read client user input messages from network
3. Execute client user input messages
4. Simulate server-controlled objects using simulation time from last full pass
5. For each connected client, package up visible objects/world state and send to client
6. Sample clock to find end time

7. End time minus start time is the simulation time for the next frame

In this model, non-player objects run purely on the server, while player objects drive their movements based on incoming packets. Of course, this is not the only possible way to accomplish this task, but it does make sense.

Contents of the User Input messages

In Half-Life engine games, the user input message format is quite simple and is encapsulated in a data structure containing just a few essential fields:

```
typedef struct usercmd_s
{
    // Interpolation time on client
    short       lerp_msec;
    // Duration in ms of command
    byte        msec;
    // Command view angles.
    vec3_t  viewangles;
    // intended velocities
    // Forward velocity.
    float       forwardmove;
    // Sideways velocity.
    float       sidemove;
    // Upward velocity.
    float       upmove;
    // Attack buttons
    unsigned short buttons;
    //
    // Additional fields omitted...
    //
} usercmd_t;
```

The critical fields here are the msec, viewangles, forward, side, and upmove, and buttons fields. The msec field corresponds to the number of milliseconds of simulation that the command corresponds to (it's the frametime). The viewangles field is a vector representing the direction the player was looking during the frame. The forward, side, and upmove fields are the impulses determined by examining the keyboard, mouse, and joystick to see if any movement keys were held down. Finally, the buttons field is just a bit field with one or more bits set for each button that is being held down.

Using the above data structures and client / server architecture, the core of the simulation is as follows. First, the client creates and sends a user command to the server. The server then executes the user command and sends updated positions of everything back to client. Finally, the client renders the scene with all of these objects. This core, though quite simple, does not react well under real world situations, where users can experience significant amounts of latency in their Internet connections. The main problem is that the client truly is "dumb" and all it does is the simple task of sampling movement inputs and waiting for the server to tell it the results. If the client has 500 milliseconds of latency in its connection to the server, then it will take 500 milliseconds for any client actions to be acknowledged by the server and for the results to be perceptible on the client. While this round trip delay may be acceptable on a Local Area Network (LAN), it is not acceptable on the Internet.

Client Side Prediction

One method for ameliorating this problem is to perform the client's movement locally and just assume, temporarily, that the server will accept and acknowledge the client commands directly. This method is labeled as client-side prediction.

Client-side prediction of movements requires us to let go of the "dumb" or minimal client principle. That's not to say that the client is fully in control of its simulation, as in a peer-to-peer game with no central server. There still is an authoritative server running the simulation just as noted above. Having an authoritative server means that even if the client simulates different results than the server, the server's results will eventually correct the client's incorrect simulation. Because of the latency in the connection, the correction might not occur until a full round trip's worth of time has passed. The downside is that this can cause a very perceptible shift in the player's position due to the fixing up of the prediction error that occurred in the past.

To implement client-side prediction of movement, the following general procedure is used. As before, client inputs are sampled and a user command is generated. Also as before, this user command is sent off to the server. However, each user command (and the exact time it was generated) is stored on the client. The prediction algorithm uses these stored commands.

For prediction, the last acknowledged movement from the server is used as a starting point. The acknowledgement indicates which user command was last acted upon by the server and also tells us the exact position (and other state data) of the player after that movement command was simulated on the server. The last acknowledged command will be somewhere in the past if there is any lag in the connection. For instance, if the client is running at 50 frames per second (fps) and has 100 milliseconds of latency (roundtrip), then the client will have stored up five user commands ahead of the last one acknowledged by the server. These five user commands are simulated on the client as a part of client-side prediction. Assuming full prediction[1], the client will want to start with the latest data from the server, and then run the five user commands through "similar logic" to what the server uses for simulation of client movement. Running these commands should produce an accurate final state on the client (final player position is most important) that can be used to determine from what position to render the scene during the current frame.

In Half-Life, minimizing discrepancies between client and server in the prediction logic is accomplished by sharing the identical movement code for players in both the server-side game code and the client-side game code. These are the routines in the `pm_shared/` (which stands for "player movement shared") folder of the HL SDK (http://download.cnet.com/downloads/0-10045-100-3422497.html). The input to the shared routines is encapsulated by the user command and a "from" player state. The output is the new player state after issuing the user command. The general algorithm on the client is as follows:

```
"from state" <- state after last user command acknowledged by the server;

"command" <- first command after last user command acknowledged by server;

while (true)
{
        run "command" on "from state" to generate "to state";
        if (this was the most up to date "command")
                break;

        "from state" = "to state";
        "command" = next "command";
};
```

The origin and other state info in the final "to state" is the prediction result and is used for rendering the scene that frame. The portion where the command is run is simply the portion where all of the player state data is copied into the shared data structure, the user command is processed (by executing the common code in the pm_shared routines in Half-Life's case), and the resulting data is copied back out to the "to state".

There are a few important caveats to this system. First, you'll notice that, depending upon the client's latency and how fast the client is generating user commands (i.e., the client's framerate), the client will most often end up running the same commands over and over again until they are finally acknowledged by the server and dropped from the list (a sliding window in Half-Life's case) of commands yet to be acknowledged. The first consideration is how to handle any sound effects and visual effects that are created in the shared code. Because commands can be run over and over again, it's important not to create footstep sounds, etc. multiple times as the old commands are re-run to update the predicted position. In addition, it's important for the server not to send the client effects that are already being predicted on the client. However, the client still must re-run the old commands or else there will be no way for the server to correct any erroneous prediction by the client. The solution to this problem is easy: the client just marks those commands which have not been predicted yet on the client and only plays effects if the user command is being run for the first time on the client.

The other caveat is with respect to state data that exists solely on the client and is not part of the authoritative update data from the server. If you don't have any of this type of data, then you can simply use the last acknowledged state from the server as a starting point, and run the prediction user commands "in-place" on that data to arrive at a final state (which includes your position for rendering). In this case, you don't need to keep all of the intermediate results along the route for predicting from the last acknowledged state to the current time. However, if you are doing any logic totally client side (this logic could include functionality such as determining where the eye position is when you are in the process of crouching—and it's not really totally client side since the server still simulates this data also) that affects fields that are not replicated from the server to the client by the networking layer handling the player's state info, then you will need to store the intermediate results of prediction. This can be done with a sliding window, where the "from state" is at the start and then each time you run a user command through prediction, you fill in the next state in the window. When the server finally acknowledges receiving one or more commands that had been predicted, it is a simple matter of looking up which state the server is acknowledging and copying over the data that is totally client side to the new starting or "from state".

So far, the above procedure describes how to accomplish client side prediction of movements. This system is similar to the system used in QuakeWorld[2].

Client-Side Prediction of Weapon Firing

Layering prediction of the firing effects of weapons onto the above system is straightforward. Additional state information is needed for the local player on the client, of course, including which weapons are being held, which one is active, and how much ammo each of these weapons has remaining. With this information, the firing logic can be layered on top of the movement logic because, once again, the state of the firing buttons is included in the user command data structure that is shared between the client and the server. Of course, this can get complicated if the actual weapon logic is different between client and server. In Half-Life, we chose to avoid this complication by moving the implementation of a weapon's firing logic into "shared code" just like the player movement code. All of the variables that contribute to determining weapon state (e.g., ammo, when the next firing of the weapon can occur, what weapon animation is playing, etc.), are then part of the authoritative server state and are replicated to the client-side so there, they can be used for prediction of weapon state.

Predicting weapon firing on the client will likely lead to the decision also to predict weapon switching, deployment, and holstering. In this fashion, the user feels that the game is 100% responsive to his or her movement and weapon activation activities. This goes a long way toward reducing the feeling of latency that many players have come to endure with today's Internet-enabled action experiences.

Umm, This is a Lot of Work

Replicating the necessary fields to the client and handling all of the intermediate state is a fair amount of work. At this point, you may be asking, why not eliminate all of the server stuff and just have the client report where s/he is after each movement? In other words, why not ditch the server stuff and just run the movement and weapons purely on the client-side? Then, the client would just send results to the server along the lines of, "I'm now at position x and, by the way, I just shot player 2 in the head." This is fine if you can trust the client. This is how a lot of the military simulation systems work (i.e., they are a closed system and they trust all of the clients). This is how peer-to-peer games generally work. For Half-Life, this mechanism is unworkable because of realistic concerns about cheating. If we encapsulated absolute state data in this fashion, we'd raise the motivation to hack the client even higher than it already is[3]. For our games, this risk is too high and we fall back to requiring an authoritative server.

A system where movements and weapon effects are predicted client-side is a very workable system. For instance, this is the system that the Quake3 engine supports. One of the problems with this system is that you still have to have a feel for your latency to determine how to lead your targets (for instant hit weapons). In other words, although you get to hear the weapons firing immediately, and your position is totally up-to-date, the results of your shots are still subject to latency. For example, if you are aiming at a player running perpendicular to your view and you have 100 milliseconds of latency and the player is running at 500 units per second, then you'll need to aim 50 units in front of the target to hit the target with an instant hit weapon. The greater the latency, the greater the lead targeting needed. Getting a "feel" for your latency is difficult. Quake3 attempted to mitigate this by playing a brief tone whenever you received confirmation of your hits. That way, you could figure out how far to lead by firing your weapons in rapid succession and adjusting your leading amount until you started to hear a steady stream of tones. Obviously, with sufficient latency and an opponent who is actively dodging, it is quite difficult to get enough feedback to focus in on the opponent in a consistent fashion. If your latency is fluctuating, it can be even harder.

Display of Targets

Another important aspect influencing how a user perceives the responsiveness of the world is the mechanism for determining, on the client, where to render the other players. The two most basic mechanisms for determining where to display objects are extrapolation and interpolation[4].

For extrapolation, the other player/object is simulated forward in time from the last known spot, direction, and velocity in more or less a ballistic manner. Thus, if you are 100 milliseconds lagged, and the last update you received was that (as above) the other player was running 500 units per second perpendicular to your view, then the client could assume that in "real time" the player has moved 50 units straight ahead from that last known position. The client could then just draw the player at that extrapolated position and the local player could still more or less aim right at the other player.

The biggest drawback of using extrapolation is that player's movements are not very ballistic, but instead are very non-deterministic and subject to high jerk[5]. Layer on top of this the unrealistic player physics models that most FPS games use, where player's can turn instantaneously and apply unrealistic forces to create huge accelerations at arbitrary angles and you'll see that the extrapolation is quite often incorrect. The developer can mitigate the error by limiting the extrapolation time to a reasonable value (QuakeWorld, for instance, limited extrapolation to 100 milliseconds). This limitation helps because, once the true player position is finally received, there will be a limited amount of corrective warping. In a world where most players still have greater than 150 milliseconds of latency, the player must still lead other players in order to hit them. If those players are "warping" to new spots because of extrapolation errors, then the gameplay suffers nonetheless.

The other method for determining where to display objects and players is interpolation. Interpolation can be viewed as always moving objects somewhat in the past with respect to the last valid position received for the object. For instance, if the server is sending 10 updates per second (exactly) of the world state, then we might impose 100 milliseconds of interpolation delay in our rendering. Then, as we render frames, we interpolate the position of the object between the last updated position and the position one update before that (alternatively, the last render position) over that 100 milliseconds. As the object just gets to the last updated position, we receive a new update from the server (since 10 updates per second means that the updates come in every 100 milliseconds) we can start moving toward this new position over the next 100 milliseconds.

If one of the update packets fails to arrive, then there are two choices: We can start extrapolating the player position as noted above (with the large potential errors noted) or we can simply have the player rest at the position in the last update until a new update arrives (causing the player's movement to stutter).

The general algorithm for this type of interpolation is as follows:

1. Each update contains the server time stamp for when it was generated[6]
2. From the current client time, the client computes a target time by subtracting the interpolation time delta (100 ms)
3. If the target time is in between the timestamp of the last update and the one before that, then those timestamps determine what fraction of the time gap has passed.
4. This fraction is used to interpolate any values (e.g., position and angles).

In essence, you can think of interpolation, in the above example, as buffering an additional 100 milliseconds of data on the client. The other players, therefore, are drawn where they were at a point in the past that is equal to your exact latency plus the amount of time over which you are interpolating. To deal with the occasional dropped packet, we could set the interpolation time as 200 milliseconds instead of 100 milliseconds. This would (again assuming 10 updates per second from the server) allow us to entirely miss one update and still have the player interpolating toward a valid position, often moving through this interpolation without a hitch. Of course, interpolating for more time is a tradeoff, because it is trading additional latency (making the interpolated player harder to hit) for visual smoothness.

In addition, the above type of interpolation (where the client tracks only the last two updates and is always moving directly toward the most recent update) requires a fixed time interval between server updates. The method also suffers from visual quality issues that are difficult to resolve. The visual quality issue is as follows. Imagine that the object being interpolated is a bouncing ball (which actually accurately describes some of our players). At the extremes, the ball is either high in the air or hitting the pavement. However, on average, the ball is somewhere in between. If we only interpolate to the last position, it is very likely that this position is not on the ground or at the high point. The bounciness of the ball is "flattened" out and it never seems to hit the ground. This is a classical sampling problem and can be alleviated by sampling the world state more frequently. However, we are still quite likely never actually to have an interpolation target state be at the ground or at the high point and this will still flatten out the positions.

In addition, because different users have different connections, forcing updates to occur at a lockstep like 10 updates per second is forcing a lowest common denominator on users unnecessarily. In Half-Life, we allow the user to ask for as many updates per second as he or she wants (within limit). Thus, a user with a fast connection could receive 50 updates per second if the user wanted. By default, Half-Life sends 20 updates per second to each player the Half-Life client interpolates players (and many other objects) over a period of 100 milliseconds.[7]

To avoid the flattening of the bouncing ball problem, we employ a different algorithm for interpolation. In this method, we keep a more complete "position history" for each object that might be interpolated.

The position history is the timestamp and origin and angles (and could include any other data we want to interpolate) for the object. Each update we receive from the server creates a new position history entry, including timestamp and origin/angles for that timestamp. To interpolate, we compute the target time as above, but then we search backward through the history of positions looking for a pair of updates that straddle the target time. We then use these to interpolate and compute the final position for that frame. This allows us to smoothly follow the curve that completely includes all of our sample points. If we are running at a higher framerate than the incoming update rate, we are almost assured of smoothly moving through the sample points, thereby minimizing (but not eliminating, of course, since the pure sampling rate of the world updates is the limiting factor) the flattening problem described above.

The only consideration we have to layer on top of either interpolation scheme is some way to determine that an object has been forcibly teleported, rather than just moving really quickly. Otherwise we might "smoothly" move the object over great distances, causing the object to look like it's traveling way too fast. We can either set a flag in the update that says, "don't interpolate" or "clear out the position history," or we can determine if the distance between the origin and one update and another is too big, and thereby presumed to be a teleportation/warp. In that case, the solution is probably to just move the object to the latest know position and start interpolating from there.

Lag Compensation

Understanding interpolation is important in designing for lag compensation because interpolation is another type of latency in a user's experience. To the extent that a player is looking at other objects that have been interpolated, then the amount of interpolation must be taken into consideration in computing, on the server, whether the player's aim was true.

Lag compensation is a method of normalizing server-side the state of the world for each player as that player's user commands are executed. You can think of lag compensation as taking a step back in time, on the server, and looking at the state of the world at the exact instant that the user performed some action. The algorithm works as follows:

1. Before executing a player's current user command, the server:
   1. Computes a fairly accurate latency for the player
   2. Searches the server history (for the current player) for the world update that was sent to the player and received by the player just before the player would have issued the movement command
   3. From that update (and the one following it based on the exact target time being used), for each player in the update, move the other players backwards in time to exactly where they were when the current player's user command was created. This moving backwards must account for both connection latency and the interpolation amount[8] the client was using that frame.
2. Allow the user command to execute (including any weapon firing commands, etc., that will run ray casts against all of the other players in their "old" positions).
3. Move all of the moved/time-warped players back to their correct/current positions

Note that in the step where we move the player backwards in time, this might actually require forcing additional state info backwards, too (for instance, whether the player was alive or dead or whether the player was ducking). The end result of lag compensation is that each local client is able to directly aim at other players without having to worry about leading his or her target in order to score a hit. Of course, this behavior is a game design tradeoff.

Game Design Implications of Lag Compensation

The introduction of lag compensation allows for each player to run on his or her own clock with no apparent latency. In this respect, it is important to understand that certain paradoxes or inconsistencies can occur. Of course, the old system with the authoritative server and "dumb" or simple clients had it's own paradoxes. In the end, making this tradeoff is a game design decision. For Half-Life, we believe deciding in favor of lag compensation was a justified game design decision.

The first problem of the old system was that you had to lead your target by some amount that was related to your latency to the server. Aiming directly at another player and pressing the fire button was almost assured to miss that player. The inconsistency here is that aiming is just not realistic and that the player controls have non-predictable responsiveness.

With lag compensation, the inconsistencies are different. For most players, all they have to do is acquire some aiming skill and they can become proficient (you still have to be able to aim). Lag compensation allows the player to aim directly at his or her target and press the fire button (for instant hit weapons[9]). The inconsistencies that sometimes occur, however, are from the points of view of the players being fired upon.

For instance, if a highly lagged player shoots at a less lagged player and scores a hit, it can appear to the less lagged player that the lagged player has somehow "shot around a corner"[10]. In this case, the lower lag player may have darted around a corner. But the lagged player is seeing everything in the past. To the lagged player, s/he has a direct line of sight to the other player. The player lines up the crosshairs and presses the fire button. In the meantime, the low lag player has run around a corner and maybe even crouched behind a crate. If the high lag player is sufficiently lagged, say 500 milliseconds or so, this scenario is quite possible. Then, when the lagged player's user command arrives at the server, the hiding player is transported backward in time and is hit. This is the extreme case, and in this case, the low ping player says that s/he was shot from around the corner. However, from the lagged player's point of view, they lined up their crosshairs on the other player and fired a direct hit. From a game design point of view, the decision for us was easy: let each individual player have completely responsive interaction with the world and his or her weapons.

In addition, the inconsistency described above is much less pronounced in normal combat situations. For first-person shooters, there are two more typical cases. First, consider two players running straight at each other pressing the fire button. In this case, it's quite likely that lag compensation will just move the other player backwards along the same line as his or her movement. The person being shot will be looking straight at his attacker and no "bullets bending around corners" feeling will be present.

The next example is two players, one aiming at the other while the other dashes in front perpendicular to the first player. In this case, the paradox is minimized for a wholly different reason. The player who is dashing across the line of sight of the shooter probably has (in first-person shooters at least) a field of view of 90 degrees or less. In essence, the runner can't see where the other player is aiming. Therefore, getting shot isn't going to be surprising or feel wrong (you get what you deserve for running around in the open like a maniac). Of course, if you have a tank game, or a game where the player can run one direction, and look another, then this scenario is less clear-cut, since you might see the other player aiming in a slightly incorrect direction.

Conclusion

Lag compensation is a tool to ameliorate the effects of latency on today's action games. The decision of whether to implement such a system rests with the game designer since the decision directly changes the feel of the game. For Half-Life, Team Fortress and Counter Strike, the benefits of lag compensation easily outweighed the inconsistencies noted above.

Footnotes

1. In the Half-Life engine, it is possible to ask the client-side prediction algorithm to account for some, but not all, of the latency in performing prediction. The user could control the amount of prediction by changing the value of the "pushlatency" console variable to the engine. This variable is a negative number indicating the maximum number of milliseconds of prediction to perform. If the number is greater (in the negative) than the user's current latency, then full prediction up to the current time occurs. In this case, the user feels zero latency in his or her movements. Based upon some erroneous superstition in the community, many users insisted that setting pushlatency to minus one-half of the current average latency was the proper setting. Of course, this would still leave the player's movements lagged (often described as if you are moving around on ice skates) by half of the user's latency. All of this confusion has brought us to the conclusion that full prediction should occur all of the time and that the pushlatency variable should be removed from the Half-Life engine. (Return)
2. http://www.quakeforge.net/files/q1source.zip (Return)

3. A discussion of cheating and what developers can do to deter it is beyond the scope of this paper. (Return)
4. Though hybrids and corrective methods are also possible. (Return)
5. "Jerk" is a measure of how fast accelerative forces are changing. (Return)
6. It is assumed in this paper that the client clock is directly synchronized to the server clock modulo the latency of the connection. In other words, the server sends the client, in each update, the value of the server's clock and the client adopts that value as its clock. Thus, the server and client clocks will always be matched, with the client running the same timing somewhat in the past (the amount in the past is equal to the client's current latency). Smoothing out discrepancies in the client clock can be solved in various ways. (Return)
7. The time spacing of these updates is not necessarily fixed. The reason why is that during high activity periods of the game (especially for users with lower bandwidth connections), it's quite possible that the game will want to send you more data than your connection can accommodate. If we were on a fixed update interval, then you might have to wait an entire additional interval before the next packet would be sent to the client. However, this doesn't match available bandwidth effectively. Instead, the server, after sending every packet to a player, determines when the next packet can be sent. This is a function of the user's bandwidth or "rate" setting and the number of updates requested per second. If the user asks for 20 updates per second, then it will be at least 50 milliseconds before the next update packet can be sent. If the bandwidth choke is active (and the server is sufficiently high framerate), it could be 61, etc., milliseconds before the next packet gets sent. Thus, Half-Life packets can be somewhat arbitrarily spaced. The simple move to latest goal interpolation schemes don't behave as well (think of the old anchor point for movement as being variable) under these conditions as the position history interpolation method (described below). (Return)
8. Which Half-Life encodes in the lerp_msec field of the usercmd_t structure described previously. (Return)
9. For weapons that fire projectiles, lag compensation is more problematic. For instance, if the projectile lives autonomously on the server, then what time space should the projectile live in? Does every other player need to be "moved backward" every time the projectile is ready to be simulated and moved by the server? If so, how far backward in time should the other players be moved? These are interesting questions to consider. In Half-Life, we avoided them; we simply don't lag compensate projectile objects (that's not to say that we don't predict the sound of you firing the projectile on the client, just that the actual projectile is not lag compensated in any way). (Return)
10. This is the phrase our user community has adopted to describe this inconsistency. (Return)
