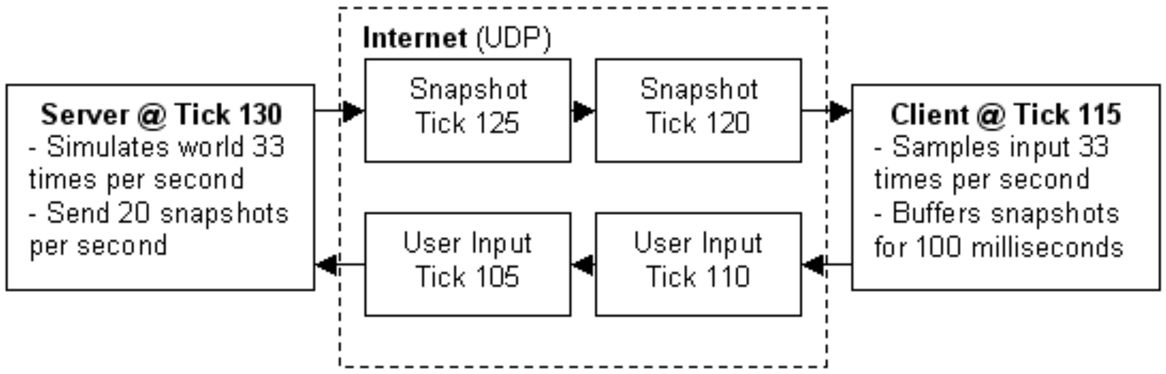
Source Multiplayer Networking

Multiplayer games based on the Source Engine use a Client-Server networking architecture. Usually a server is a dedicated host that runs the game and is authoritative about world simulation, game rules, and player input processing. A client is a player's computer connected to a game server. The client and server communicate with each other by sending small data packets at a high frequency (usually 20 to 30 packets per second). A client receives the current world state from the server and generates video and audio output based on these updates. The client also samples data from input devices (keyboard, mouse, microphone, etc.) and sends these input samples back to the server for further processing. Clients only communicate with the game server and not between each other (like in a peer-to-peer application). In contrast with a single player game, a multiplayer game has to deal with a variety of new problems caused by packet-based communication.

Network bandwidth is limited, so the server can't send a new update packet to all clients for every single world change. Instead, the server takes snapshots of the current world state at a constant rate and broadcasts these snapshots to the clients. Network packets take a certain amount of time to travel between the client and the server (i.e. half the ping time). This means that the client time is always a little bit behind the server time. Furthermore, client input packets are also delayed on their way back, so the server is processing temporally delayed user commands. In addition, each client has a different network delay which varies over time due to other background traffic and the client's framerate. These time differences between server and client causes logical problems, becoming worse with increasing network latencies. In fast-paced action games, even a delay of a few milliseconds can cause a laggy gameplay feeling and make it hard to hit other players or interact with moving objects. Besides bandwidth limitations and network latencies, information can get lost due to network packet loss.


Internet (UDP)
Server @ Tick 130
- Simulates world 33 times per second
- Send 20 snapshots per second
Snapshot Tick 125
Snapshot Tick 120
Client @ Tick 115
- Samples input 33 times per second
- Buffers snapshots for 100 milliseconds
User Input Tick 105
User Input Tick 110

To cope with the issues introduced by network communication, the Source engine server employs techniques such as data compression and lag compensation which are invisible to the client. The client then performs prediction and interpolation to further improve the experience.

Basic networking

The server simulates the game in discrete time steps called ticks. By default, the timestep is 15ms, so 66.666... ticks per second are simulated, but mods can specify their own tickrate. During each tick, the server processes incoming user commands, runs a physical simulation step, checks the game rules, and updates all object states. After simulating a tick, the server decides if any client needs a world update and takes a snapshot of the current world state if necessary. A higher tickrate increases the simulation precision, but also requires more CPU power and available bandwidth on both server and client. The server admin may override the default tickrate with the `-tickrate` command line parameter, though tickrate changes done this way are not recommended because the mod may not work as designed if its tickrate is changed.

Clients usually have only a limited amount of available bandwidth. In the worst case, players with a modem connection can't receive more than 5 to 7 KB/sec. If the server tried to send them updates with a higher data rate, packet loss would be unavoidable. Therefore, the client has to tell the server its incoming bandwidth capacity by setting the console variable `rate` (in bytes/second). This is the most important network variable for clients and it has to be set correctly for an optimal gameplay experience. The client can request a certain snapshot rate by changing `cl_updaterate` (default 20), but the server will never send more updates than simulated ticks or exceed the requested client `rate` limit. Server admins can limit data rate values requested by clients with `sv_minrate` and `sv_maxrate` (both in bytes/second). Also the snapshot rate can be restricted with `sv_minupdaterate` and `sv_maxupdaterate` (both in snapshots/second).

The client creates *user commands* from sampling input devices with the same tick rate that the server is running with. A user command is basically a snapshot of the current keyboard and mouse state. But instead of sending a new packet to the server for each user command, the client sends command packets at a certain rate of packets per second (usually 30). This means two or more user commands are transmitted within the same packet. Clients can increase the command rate with `cl_cmdrate`. This will increase responsiveness but requires more outgoing bandwidth, too.

Game data is compressed using *delta compression* to reduce network load. That means the server doesn't send a full world snapshot each time, but rather only changes (a delta snapshot) that happened since the last acknowledged update. With each packet sent between the client and server, acknowledge numbers are attached to keep track of their data flow. Usually full (non-delta) snapshots are only sent when a game starts or a client suffers from heavy packet loss for a couple of seconds. Clients can request a full snapshot manually with the `cl_fullupdate` command.

Responsiveness, or the time between user input and its visible feedback in the game world, are determined by lots of factors, including the server/client CPU load, simulation tickrate, data rate and snapshot update settings, but mostly by the network packet traveling time. The time between the client sending a user command, the server responding to it, and the client receiving the server's response is called the *latency* or *ping* (or round trip time). Low latency is a significant advantage when playing a multiplayer online game. Techniques like prediction and lag compensation try to minimize that advantage and allow a fair game for players with slower connections. Tweaking networking setting can help to gain a better experience if the necessary bandwidth and CPU power is available. We recommend keeping the default settings, since improper changes may cause more negative side effects than actual benefits.

Servers that Support Tickrate

The tickrate can be altered by using the `-tickrate` parameter

- Counter Strike: Global Offensive
- Half-Life 2: Deathmatch

The following servers tickrate cannot be altered as changing this causes server timing issues.

Tickrate 66

- Counter Strike: Source
- Day of Defeat: Source
- Team Fortress 2

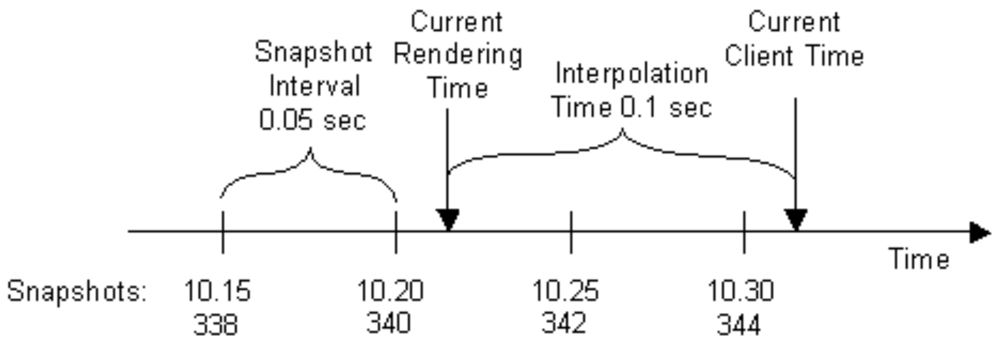
Tickrate 30

- Left 4 Dead
- Left 4 Dead 2

Entity interpolation

By default, the client receives about 20 snapshot per second. If the objects (entities) in the world were only rendered at the positions received by the server, moving objects and animation would look choppy and jittery. Dropped packets would also cause noticeable glitches. The trick to solve this problem is to go back in time for rendering, so positions and animations can be continuously interpolated between two recently received snapshots. With 20 snapshots per second, a new update arrives about every 50 milliseconds. If the client render time is shifted back by 50 milliseconds, entities can be always interpolated between the last received snapshot and the snapshot before that.

Source defaults to an interpolation period ('lerp') of 100-milliseconds (`cl_interp 0.1`); this way, even if one snapshot is lost, there are always two valid snapshots to interpolate between. Take a look at the following figure showing the arrival times of incoming world snapshots:



The last snapshot received on the client was at tick 344 or 10.30 seconds. The client time continues to increase based on this snapshot and the client frame rate. If a new video frame is rendered, the rendering time is the current client time 10.32 minus the view interpolation delay of 0.1 seconds. This would be 10.22 in our example and all entities and their animations are interpolated using the correct fraction between snapshot 340 and 342.

Since we have an interpolation delay of 100 milliseconds, the interpolation would even work if snapshot 342 were missing due to packet loss. Then the interpolation could use snapshots 340 and 344. If more than one snapshot in a row is dropped, interpolation can't work perfectly because it runs out of snapshots in the history buffer. In that case the renderer uses extrapolation (`cl_extrapolate 1`) and tries a simple linear extrapolation of entities based on their known history so far. The extrapolation is done only for 0.25 seconds of packet loss (`cl_extrapolate_amount`), since the prediction errors would become too big after that.

Entity interpolation causes a constant view "lag" of 100 milliseconds by default (`cl_interp 0.1`), even if you're playing on a listenserver (server and client on the same machine). This doesn't mean you have to lead your aiming when shooting at other players since the server-side lag compensation knows about client entity interpolation and corrects this error.

Input prediction

Lets assume a player has a network latency of 150 milliseconds and starts to move forward. The information that the `+FORWARD` key is pressed is stored in a user command and send to the server. There the user command is processed by the movement code and the player's character is moved forward in the game world. This world state change is transmitted to all clients with the next snapshot update. So the player would see his own change of movement with a 150 milliseconds delay after he started walking. This delay applies to all players actions like movement, shooting weapons, etc. and becomes worse with higher latencies.

A delay between player input and corresponding visual feedback creates a strange, unnatural feeling and makes it hard to move or aim precisely. Client-side input prediction (`cl_predict 1`) is a way to remove this delay and let the player's actions feel more instant. Instead of waiting for the server to update your own position, the local client just predicts the results of its own user commands. Therefore, the client runs exactly the same code and rules the server will use to process the user commands. After the prediction is finished, the local player will move instantly to the new location while the server still sees him at the old place.

After 150 milliseconds, the client will receive the server snapshot that contains the changes based on the user command he predicted earlier. Then the client compares the server position with his predicted position. If they are different, a prediction error has occurred. This indicates that the client didn't have the correct information about other entities and the environment when it processed the user command. Then the client has to correct its own position, since the server has final authority over client-side prediction. If `cl_showerror 1` is turned on, clients can see when prediction errors happen. Prediction error correction can be quite noticeable and may cause the client's view to jump erratically. By gradually correcting this error over a short amount of time (`cl_smoothtime`), errors can be smoothly corrected. Prediction error smoothing can be turned off with `cl_smooth 0`.

Prediction is only possible for the local player and entities affected only by him, since prediction works by using the client's keypresses to make a "best guess" of where the player will end up. Predicting other players would require literally predicting the future with no data, since there's no way to instantaneously get keypresses from them.

Lag compensation

> *All source code for lag compensation and view interpolation is available in the Source SDK. See Lag compensation for implementation details.*

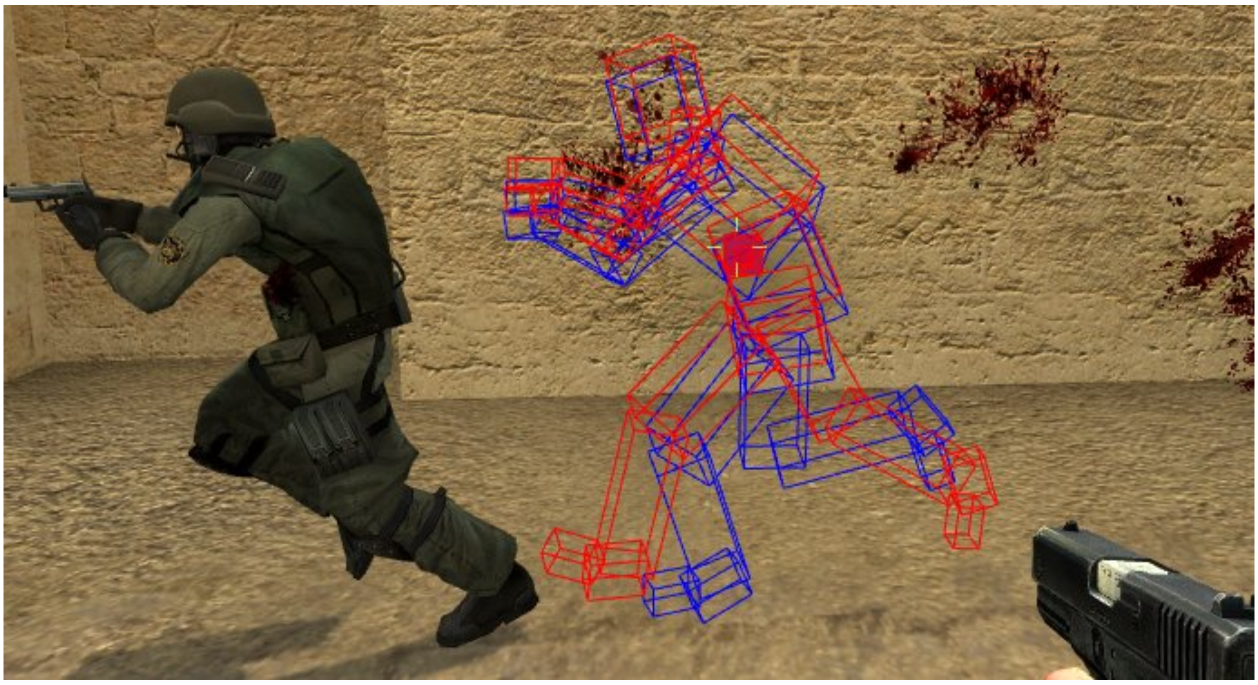
Let's say a player shoots at a target at client time 10.5. The firing information is packed into a user command and sent to the server. While the packet is on its way through the network, the server continues to simulate the world, and the target might have moved to a different position. The user command arrives at server time 10.6 and the server wouldn't detect the hit, even though the player has aimed exactly at the target. This error is corrected by the server-side lag compensation.

The lag compensation system keeps a history of all recent player positions for one second. If a user command is executed, the server estimates at what time the command was created as follows:

Command Execution Time = **Current Server Time** - **Packet Latency** - **Client View Interpolation**

Then the server moves all other players - *only* players - back to where they were at the command execution time. The user command is executed and the hit is detected correctly. After the user command has been processed, the players revert to their original positions.

On a listen server you can enable `sv_showimpacts 1` to see the different server and client hitboxes:



This screenshot was taken on a listen server with 200 milliseconds of lag (using `net_fakelag`), right after the server confirmed the hit. The red hitbox shows the target position on the client where it was 100ms + interp period ago. Since then, the target continued to move to the left while the user command was travelling to the server. After the user command arrived, the server restored the target position (blue hitbox) based on the estimated command execution time. The server traces the shot and confirms the hit (the client sees blood effects).

Client and server hitboxes don't exactly match because of small precision errors in time measurement. Even a small difference of a few milliseconds can cause an error of several inches for fast-moving objects. Multiplayer hit detection is not pixel perfect and has known precision limitations based on the tickrate and the speed of moving objects.

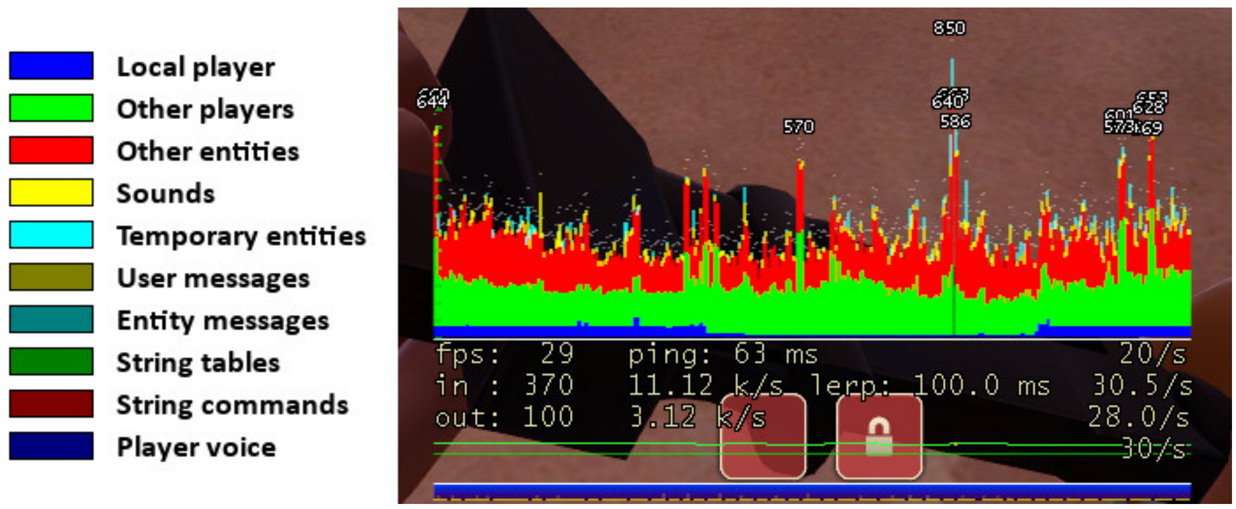
The question arises, why is hit detection so complicated on the server? Doing the back tracking of player positions and dealing with precision errors while hit detection could be done client-side way easier and with pixel precision. The client would just tell the server with a "hit" message what player has been hit and where. We can't allow that simply because a game server can't trust the clients on such important decisions. Even if the client is "clean" and protected by Valve Anti-Cheat, the packets could be still modified on a 3rd machine while routed to the game server. These "cheat proxies" could inject "hit" messages into the network packet without being detected by VAC (a "man-in-the-middle" attack).

Network latencies and lag compensation can create paradoxes that seem illogical compared to the real world. For example, you can be hit by an attacker you can't even see anymore because you already took cover. What happened is that the server moved your player hitboxes back in time, where you were still exposed to your attacker. This inconsistency problem can't be solved in general because of the relatively slow packet speeds. In the real world, you don't notice this problem because light (the packets) travels so fast and you and everybody around you sees the same world as it is right now.

Net graph

The Source engine offers a couple of tools to check your client connection speed and quality. The most popular one is the net graph, which can be enabled with `net_graph 2` (or `+graph`). Incoming packets are represented by small lines moving from right to left. The height of each line reflects size of a packet. If a gap appears between lines, a packet was lost or arrived out of order. The lines are color-coded depending on what kind of data they contain.

Under the net graph, the first line shows your current rendered frames per second, your average latency, and the current value of `cl_updaterate`. The second line shows the size in bytes of the last incoming packet (snapshots), the average incoming bandwidth, and received packets per second. The third line shows the same data just for outgoing packets (user commands).



```
fps:   29    ping: 63 ms              20/s
in :  370    11.12 k/s  lerp: 100.0 ms  30.5/s
out:  100    3.12 k/s                 28.0/s
                                      30/s
```

Optimizations

The default networking settings are designed for playing on dedicated server on the Internet. The settings are balanced to work well for most client/server hardware and network configurations. For Internet games the only console variable that should be adjusted on the client is "rate", which defines your available bytes/second bandwidth of your network connection. Good values for "rate" is 4500 for modems, 6000 for ISDN, 10000 DSL and above.

In an high-performance network environment, where the server and all clients have the necessary hardware resources available, it's possible to tweak bandwidth and tickrate settings to gain more gameplay precision. Increasing the server tickrate generally improves movement and shooting precision but comes with a higher CPU cost. A Source server running with tickrate 100 generates about 1.5x more CPU load than a default tickrate 66. That can cause serious calculation lags, especially when lots of people are shooting at the same time. It's not suggested to run a game server with a higher tickrate than 66 to reserve necessary CPU resources for critical situations.

If the game server is running with a higher tickrate, clients can increase their snapshot update rate (cl_updaterate) and user command rate (cl_cmdrate), if the necessary bandwidth (rate) is available. The snapshot update rate is limited by the server tickrate, a server can't send more then one update per tick. So for a tickrate 66 server, the highest client value for cl_updaterate would be 66. If you increase the snapshot rate and encounter packet loss or choke, you have to turn it down again. With an increased cl_updaterate you can also lower the view interpolation delay (cl_interp). The default interpolation delay is 0.1 seconds, which derives from the default cl_updaterate 20. View interpolation delay gives a moving player a small advantage over a stationary player since the moving player can see his target a split second earlier. This effect is unavoidable, but it can be reduced by decreasing the view interpolation delay. If both players are moving, the view lag delay is affecting both players and nobody has an advantage.

This is the relation between snapshot rate and view interpolation delay is the following:

interpolation period = max(cl_interp, cl_interp_ratio / cl_updaterate)

"Max(x,y)" means "whichever of these is higher". You can set `cl_interp` to 0 and still have a safe amount of interp. You can then increase cl_updaterate to decrease your interp period further, but don't exceed tickrate (66) or flood your connection with more data than it can handle.

Tips

Don't change console settings unless you are 100% sure what you are doing

Most "high-performance" setting cause exactly the opposite effect, if the server or network can't handle the load.

Don't turn off view interpolation and/or lag compensation

It will not improve movement or shooting precision.

Optimized setting for one client may not work for other clients

Do not just use settings from other clients without verifing them for your system.

If you follow a player in "First-Person" as a spectator in a game or SourceTV, you don't exactly see what the player sees

Spectators see the game world without lag compensation.

See also

- Prediction, Interpolation, Lag compensation
- Latency Compensating Methods in Client/Server In-game Protocol Design and Optimization
- TF2 Network Graph
- Networking Entities (for programmers)
- Wikipedia:Lag (online gaming)