What kind of issues are there in implementing realtime multiplayer games

Asked 16 years, 3 months ago Modified 12 years, 1 month ago

Viewed 4k times

16

I have some experience making multiplayer **turn-based** games using sockets, but I've never attempted a realtime action game. What kind of extra issues would I have to deal with? Do I need to keep a history of player actions in case lagged players do something in the past? Do I really need to use UDP packets or will TCP suffice? What else?

I haven't really decided what to make, but for the purpose of this question you can consider a 10-player 2D game with X Y movement.

sockets network-programming tcp udp multiplayer

Share

Improve this question

Follow

edited Nov 22, 2012 at 13:58

Mike Pennington

43k • 21 • 139 • 188

asked Sep 18, 2008 at 6:05

Bemmu

18.2k • 16 • 79 • 94

5 Answers

Sorted by: Highest score (default)

20

- 'client server' or 'peer to peer' or something in between: which computer has authority over which game actions.

With turn based games, normally it's very easy to just say 'the server has ultimate authority and we're done'. With real time games, often that design is a great place to start, but as soon as you add latency the client movement/actions feels unresponsive. So you add some sort of 'latency hiding' allowing the clients input to affect their character or units immediately to solve that problem, and now you have to deal with reconciling issues when the client and servers gamestate starts to diverge. 9 times outta 10 that just fine, you pop or lerp the objects the client has affected over to the authoritative position, but that 1 out of 10 times is when the object is the player avatar or something, that solution is unacceptable, so you start give the client authority over some actions. Now you have to reconcile the multiple gamestates on the server, and open yourself up to a potentially 'cheating' via a malicious client, if you care about that sort of thing. This is basically where every teleport/dupe/whatever bug/cheat comes up.

Of course you could start with a model where 'every client has authority over 'their' objects' and ignore the cheating problem (fine in quite a few cases). But now you're vulnerable to a massive affect on the game simulation if

that client drops out, or even 'just falls a little behind in keeping up with the simulation' - effectively every players game will end up being/feeling the effects of a lagging or otherwise underperforming client, in the form of either waiting for lagging client to catch up, or having the gamestate they control out of sync.

- 'synchronized' or 'asynchronus'

A common strategy to ensure all players are operating on the same gamestate is to simply agree on the list of player inputs (via one of the models described above) and then have the gameplay simulation play out synchronously on all machines. This means the simulation logic has to match exactly, or the games will go out of sync. This is actually both easier and harder than it sounds. It's easier because a game is just code, and code pretty much executes exactly the same when it's give the same input (even random number generators). It's harder because there are two cases where that's not the case: (1) when you accidently use random outside of your game simulation and (2) when you use floats. The former is rectified by having strict rules/assertions over what RNGs are use by what game systems. The latter is solved by not using floats. (floats actually have 2 problems, one they work very differently based on optimization configuration of your project, but even if that was worked out, they work inconsistently across different processor architectures atm, lol). Starcraft/Warcraft and any game that offers a 'replay' most likely use this model.

In fact, having a replay system is a great way to test that your RNGs are staying in sync.

With an asynchronus solution the game state authorities simply broadcast that entire state to all the other clients at some frequency. The clients take that data and slam that into their gamestate (and normaly do some simplistic extrapolation until they get the next update). Here's where 'udp' becomes a viable option, because you are spamming the entire gamestate every ~1sec or so, dropping some fraction of those updates is irrelevant. For games that have relatively little game state (quake, world of warcraft) this is often the simplest solution.

Share Improve this answer

Follow

answered Sep 18, 2008 at 7:51

Jeff

1,053 ● 1 ● 8 ● 14

7

Planning is your best friend. Figure out what your needs truly are.

Loading Data: Is every computer going to have the same models and graphics, and just names and locations are moved over the net. If every player can customize their character or other items, you will have to move this data around.

Cheating: do you have to worry about it? Can you trust what each client is saying. If not then you server side logic will look different than you client side logic. Imagine this simple case, each of your 10 players may have a

different movement speed because of power ups. To minimize cheating you should to calculate how far each player can move between communication updates from the server, otherwise a player could hack there speed up and nothing would stop them. If a player is consistently a little faster than expected or has a one time jump, the server would just reposition them in the closest location that was possible, because it is likely clock skew or a one time interruption in communications. However if a player is constantly moving twice as far as possible then it may be prudent to kick them out of the game. The more math, the more parts of the game state you can double check on the server, the more consistent the game will be, incidentally this will make cheating harder.

How peer to peer is it: Even if the game is going to be peer to peer you will probably want to have one player start a game and use them as a server, this is much easier than trying to manage some of the more cloud based approaches. If there is no server then you need to work a protocol for solving disputes between 2 machines with inconsistent game states.

Again planning is your best friend Plan, Plan, Plan. If you think about a problem enough you can think your way through most of the problems. Then you can start thinking about the ones you haven't solved yet.

answered Sep 18, 2008 at 7:04

Sqeaky

1,936 3 22 42

There are a few factors involved in setting up multiplayer

1. The protocol, it's important that you decide whether you want TCP or UDP. UDP has less overhead but isn't guaranteed delivery. Conversely TCP is more trustworthy. Each game will have their preferred protocol. UDP for instance will work for a first person shooter but may not be suited for an RTS where information needs to be consistent
2. Firewall/Connection. Making sure your multiplayer game doesn't have to make 2000 outbound connections and uses a standard port so portforwarding is easy. Interfacing it with windows firewall will probably be an added bonus.
3. Bandwidth. This is important, how much data are you intending to push through a network connection? I guess this will come down to play testing and recording throughput. If you're requiring upwards of 200kb/s for each client you may want to rethink a few things.
4. Server Load. This is also important, how much processing is required by a server for a normal game? Do you need some super 8 core server with 16gb of RAM to run it? Are there ways of reducing it?

I guess there are heaps more, but really you want a game that is comfortable to play over the network and over a variety of connections.

Share Improve this answer

Follow

answered Sep 18, 2008 at 6:10

Cetra

2,611 ● 1 ● 22 ● 27

I disagree with point 2, on server side I discourage at all the use of windows. – user2342558 Oct 26, 2023 at 12:35

4

How important is avoiding cheating ? [Can you trust any information coming from a client or can they be trusted and authenticated ?]

Object model How are objects communicated from one machine to another ? How are actions carried out on an object ?

Are you doing client/server or peer to peer ?

Random Numbers If you do a peer to peer then you need to keep them lock-stepped and the random numbers synchronized.

If you are doing client/server how do you deal with lag ? [dead reckoning ?]

There are a lot of non-trivial problems involved in network coding.

Check out RakNet both it's free to download code and it's discussion groups.

Share Improve this answer

answered Sep 18, 2008 at 6:46

Follow

Colin McLaughlan
41 ● 1

0

TCP is fine if your run on a LAN. But if you want to play online, you must use UDP and implement your own TCP-like layer: it's necessary to pass throw NAT routers.

You need to choose between Peer-to-peer or Client-Server communication. In Client-Server model, synchronisation and state of the world are easier to implement, but you might have a lack of reactivity online. In Pee-to-peer it's more complicated, but faster for the player.

Don't keep history of player action for game purpose (do it, but only for replay functionality). If you reach a point where it is necessary, prefer make every player wait.

Share Improve this answer

Follow

answered Sep 18, 2008 at 7:24

user12933
9 ● 1