Why can't I send UDP packets from a browser?

A solution for enabling UDP in the web

Posted by Glenn Fiedler (http://web.archive.org/web/20181107181507/https://gafferongames.com/about) on Sunday, February 26, 2017

Premise

In 2017 the most popular web games like agar.io (http://web.archive.org/web/20181107181507/https://agar.io/) are networked via WebSockets over TCP. If a UDP equivalent of WebSockets could be incorporated into browsers, it would greatly improve the networking of these games.

Background

Web browsers are built on top of HTTP, which is a stateless request/response protocol initially designed for serving static web pages. HTTP is built on top of TCP, a low-level protocol which guarantees data sent over the internet arrives reliably, and in the same order it was sent.

This has worked well for many years, but recently websites have become more interactive and poorly suited to the HTTP request/response paradigm. Rising to this challenge are modern web protocols like WebSockets, WebRTC, HTTP 2.0 and QUIC, which hold the potential to greatly improve the interactivity of the web.

Unfortunately, this new set of standards for web development don't provide what multiplayer games need, or, provide it in a form that is too complicated for game developers to use.

This leads to frustration from game developers, who just want to be able to send and receive UDP packets in the browser.

The Problem

To deliver data reliably and in order under packet loss, it is necessary for TCP to hold more recent data in a queue while waiting for dropped packets to be resent. Otherwise, data would be delivered out of order.

This is called **head of line blocking** and it creates a frustrating and almost comedically tragic problem for game developers. The most recent data they want is delayed while waiting for old data to be resent, but by the time the resent data arrives, it's too old to be used.

Unfortunately, there is no way to fix this behavior under TCP. All data must be received reliably and in order. Therefore, the standard solution in the game industry for the past 20 years has been to send game data over UDP instead.

How this works in practice is that each game develops their own custom protocol on top of UDP, implementing basic reliability as required, while sending the majority of data as unreliable-unordered. This ensures that time series data arrives as quickly as possible without waiting for dropped packets to be resent.

So, what does this have to do with web games?

The main problem for web games today is that game developers have no way to follow this industry best practice in the browser. Instead, web games send their game data over TCP, causing hitches and non-responsiveness due to head of line blocking.

This is completely unnecessary and could be fixed overnight if web games had some way to send and receive UDP packets.

What about WebSockets?

WebSockets are an extension to the HTTP protocol which upgrade a HTTP connection so that data can be exchanged bidirectionally, rather than in the traditional request/response pattern.

This elegantly solves the problem of websites that need to display dynamically changing content, because once a web socket connection is established, the server can push data to the browser without a corresponding request.

Unfortunately, since WebSockets are implemented on top of TCP, data is still subject to head of line blocking.

What about QUIC?

A key feature of QUIC is support for multiple data streams. New data streams can be created implicitly by the client or server by increasing the channel id.

The channel concept provide two key benefits:

1. Avoids a connection handshake each time a new request is made.
2. Eliminates head of line blocking between unrelated streams of data.

Unfortunately, while head of line blocking is eliminated across unrelated data streams, it still exists *within* each stream.

What about WebRTC?

WebRTC is a collection of protocols that enable peer-to-peer communication between browsers for applications like audio and video streaming.

Almost as a footnote, WebRTC supports a data channel which can be configured in unreliable mode, providing a way to send and receive unreliable-unordered data from the browser.

So why are browser games still using WebSockets in 2017?

The reason is that there is a trend away from peer-to-peer towards client/server for multiplayer games and while WebRTC makes it easy to send unreliable-unordered data from one browser to another, it falls down when data needs to be sent between a browser and a dedicated server.

It falls down because WebRTC is *extremely complex*. This complexity is understandable, being designed primarily to support peer-to-peer communication between browsers, WebRTC needs STUN, ICE and TURN support for NAT traversal and packet forwarding in the worst case.

But from a game developer point of view, all this complexity seems like dead weight, when STUN, ICE and TURN are completely completely unnecessary to communicate with dedicated servers, which have public IPs.

> ***"I feel what is needed is a UDP version of WebSockets. That's all I wish we had."*** *Matheus Valadares, creator of agar.io*

Why not just let people send UDP?

The final option to consider is to just let users send and receive UDP packets directly from the browser. Of course, this is an *absolutely terrible idea* and there are good reasons why it should never be allowed.

1. Websites would be able to launch DDoS attacks by coordinating UDP packet floods from browsers.

2. New security holes would be created as JavaScript running in web pages could craft malicious UDP packets to probe the internals of corporate networks and report back over HTTPS.

3. UDP packets are not encrypted, so any data sent over these packets could be sniffed and read by an attacker, or even modified in transmit. It would be a massive step back for web security to create a new way for browsers to send unencrypted packets.

4. There is no authentication, so a dedicated server reading packets sent from a browser would have to implement its own method to ensure that only valid clients are allowed to connect to it, which is well outside the amount of effort most game developers would be willing to apply to this problem.

So clearly, just letting JavaScript create UDP sockets in the browser is a no go.

What could a solution look like?

But what if we approach it from the other side. What if, instead of trying to bridge from the web world to games, we started with what games need and worked back to something that could work well on the web?

I'm Glenn Fiedler (http://web.archive.org/web/20181107181507/https://www.linkedin.com/in/glennfiedler) and I've been a game developer for the last 15 years. For most of this time I've specialized as a network programmer. I've got a lot of experience working on fast-paced action games. The last game I worked on was Titanfall 2 (http://web.archive.org/web/20181107181507/https://www.titanfall.com/)

About a month ago, I read this thread on Hacker News:

id=13264952)

Where the creator of agar.io (http://web.archive.org/web/20181107181507/https://agar.io/), Matheus Valadares, explained that WebRTC was too complex for him to use, and that he's still using WebSockets for his games.

I got to thinking, surely a solution must exist that's simpler than WebRTC?

I wondered what exactly this solution would look like?

My conclusion was that any solution must have these properties:

1. **Connection based** so it could not be used in DDoS attacks or to probe security holes.

2. **Encrypted** because no game or web application would want to send unencrypted packets in 2017.

3. **Authenticated** because dedicated servers only want to accept connections from clients who are authenticated on the web backend.

I would now like to present the solution. I'm not holding my breath that this would be accepted as a standard in browsers as-is, I'm a game guy, not a web guy. But I do hope at least that it will help browser creators and web developers see what client/server games actually need, and in some small way, do its part to help bridge the gap.

Hopefully the result will be multiplayer games playing better in a browser in the near future.

netcode.io

The solution I came up with is netcode.io (http://web.archive.org/web/20181107181507/http://netcode.io/)

netcode.io is a simple network protocol that lets clients securely connect to dedicated servers and communicate over UDP. It's connection oriented and encrypts and signs packets, and provides authentication support so only authenticated clients can connect to dedicated servers.

It's designed for games like agar.io (http://web.archive.org/web/20181107181507/https://agar.io/) that need to shunt players off from the main website to a number of dedicated server instances, each with some maximum number of players (up to 256 players per-instance in the reference implementation).

part of the connection handshake over UDP.

Connect tokens are short lived and rely on a shared private key between the web backend and the dedicated server instances. The benefit of this approach is that only authenticated clients are able to connect to the dedicated servers.

Where netcode.io wins out over WebRTC is simplicity. By focusing only on the dedicated server case, it removes the need for ICE, STUN and TURN. By implementing encryption, signing and authentication with libsodium (http://web.archive.org/web/20181107181507/https://libsodium.org/) it avoids the complexity of a full implementation of DTLS, while still providing the same level of security.

Over the past month I've created a reference implementation (http://web.archive.org/web/20181107181507/http://netcode.io/) of netcode.io in C. It's licenced under the BSD 3-Clause open source licence. Over the next few months, I hope to continue refining this implementation, spend time writing a spec, and work with people to port netcode.io to different languages.

Your feedback on the reference implementation is appreciated.

How it works

A client authenticates with the web backend using standard authentication techniques (eg. OAuth). Once a client is authenticated they request to play a game by making a REST call. The REST call returns a *connect token* to that client encoded as base64 over HTTPS.

A connect token has two parts:

1. A private portion, encrypted and signed by the shared private key using an AEAD primitive from libsodium. This cannot be read, modified or forged by the client.

2. A public portion, which provides information the client needs to connect, like encryption keys for UDP packets and the list of server addresses to connect to, along with some other information corresponding to the 'associated data' portion of the AEAD.

The client reads the connect token and has a list of n IP addresses to connect to in order. While n can be 1, it's best to give the client multiple servers in case the first server is full by the time the client attempts to connect to it.

additional data for the AEAD such as the netcode.io version info, protocol id (a 64bit number unique to this particular game), expiry timestamp for the connnect token and the sequence number for the AEAD primitive.

When the dedicated server receives a connection request over UDP it first checks that the contents of the packet are valid using the AEAD primitive. If any of the public data in the connection request packet is modified, the signature check will fail. This stops clients from modifying the expiry timestamp for a connect token, while also making rejection of expired tokens very fast.

Provided the connect token is valid, it is decrypted. Internally it contains a list of dedicated server addresses that the connect token is valid for, stopping malicious clients going wide with one connect token and using it to connect to all available dedicated servers.

The server also checks if the connect token has already been used by searching a short history of connect token HMACs, and ignores the connection request if a match is found. This prevents one connect token from being used to connect multiple clients.

Additionally, the server enforces that only one client with a given IP address and port may be connected at any time, and only one client by unique *client id* may be connected at any time, where *client id* is a 64 bit integer that uniquely identifies a client that has been authenticated by the web backend.

Provided the connect token has not expired, it decrypts successfully, and the dedicated server's public IP is in the list of server addresses, and all other checks pass, the dedicated server sets up a mapping between the client IP address and the encryption keys contained in the private connect token data.

All packets exchanged between the client and server from this point are encrypted using these keys. This encryption mapping expires if no UDP packets are received from the address for a short amount of time like 5 seconds.

Next, the server checks if there is room for the client on the server. Each server supports some maximum number of clients, for example a 64 player game has 64 slots for clients to connect to. If the server is full, it responds with a *connection request denied packet*. This lets clients quickly know to move on to the next server in the list when a server is full.

If there *is* room for the client, the server doesn't yet assign the client to that slot, but instead stores the address + HMAC for the connect token for that client as a *potential client*. The server then responds with a *connection challenge packet*, which contains a *challenge token* which is a block of data encrypted with a random key rolled when the server is started.

coordinate). Also, the connection challenge packet is significantly smaller than the connection request packet by design, to eliminate the possibility of the protocol being used as part of a DDoS amplification attack.

The client receives the *connection challenge packet* over UDP and switches to a state where it sends *connection response packets* to the server. Connection response packets simply reflect the *challenge token* back to the dedicated server, establishing that the client is actually able to receive packets on the source IP address they claim they are sending packets from. This stops clients with spoofed packet source addresses from connecting.

When the server receives a *connection response packet* it looks for a matching pending client entry, and if one exists, it searches once again for a free slot for the client to connect to. If there isn't one, it replies with a *connection request denied packet* since there may have been a slot free when the connection request was first received that is no longer available.

Alternatively, the server assigns the client to a free slot and replies back with a *connection keep-alive* packet, which tells the client which slot it was assigned on the server. This is known as a *client index*. In multiplayer games, this is typically used to identify clients connected to a server. For example, clients 0,1,2,3 in a 4 player game correspond to players 1,2,3 and 4.

The server now considers the client connected and is able to send *connection payload packets* down to that client. These packets wrap game specific data and are delivered unreliable-ordered. The only caveat is that since the client needs to first receive a *connection keep-alive* before it knows its client index and considers itself to be fully connected, the server tracks on a per-client slot basis whether that client is *confirmed*.

The confirmed flag per-client is initially set to false, and flips true once the server has received a keep-alive or payload packet from that client. Until a client is confirmed, each time a payload packet is sent from the server to that client, it is prefixed with a keep-alive packet. This ensures the client is statistically likely to know its client index and be fully connected prior to receiving the first payload packet sent from the server, minimizing the number of connection establishment round-trips.

Now that the client and server are fully connected they can exchange UDP packets bidirectionally. Typical game protocols sent player inputs from client to server at a high rate like 60 times per-second, and world state from the server to client at a slightly lower rate, like 20 times per-second. However more recent AAA games are increasing the server update rate.

If the server or client don't exchange a steady stream of packets, keep-alive packets are automatically generated so the connection doesn't time out. If no packets are received from either side of the connection for a short amount of time like 5 seconds, the connection times

If either side of the connection wishes to cleanly disconnect, a number of *connection disconnect packets* are fired across redundantly, so that statistically these packets are likely to get through even under packet loss. This ensures that clean disconnects happen quickly, without the other side waiting for time out.

Conclusion

Popular web games like agar.io (http://web.archive.org/web/20181107181507/https://agar.io/) are networked via WebSockets over TCP, because WebRTC is difficult to use in a client/server context with dedicated servers.

One solution would be for Google to make it *significantly* easier for game developers to integrate WebRTC data channel support in their dedicated servers.

Alternatively, netcode.io (http://web.archive.org/web/20181107181507/http://netcode.io/) provides a much simpler 'WebSockets for UDP'-like approach, which would also solve the problem, if it were standardized and incorporated into browsers.

UPDATE: netcode.io is now available in browsers (http://web.archive.org/web/20181107181507/https://github.com/RedpointGames/netcode.io-browser)!

● (http://web.archive.org/web/20181107181507/https://www.linkedin.com/in/glennfiedler/)
● (http://web.archive.org/web/20181107181507/https://twitter.com/gafferongames)
● (http://web.archive.org/web/20181107181507/https://github.com/gafferongames)