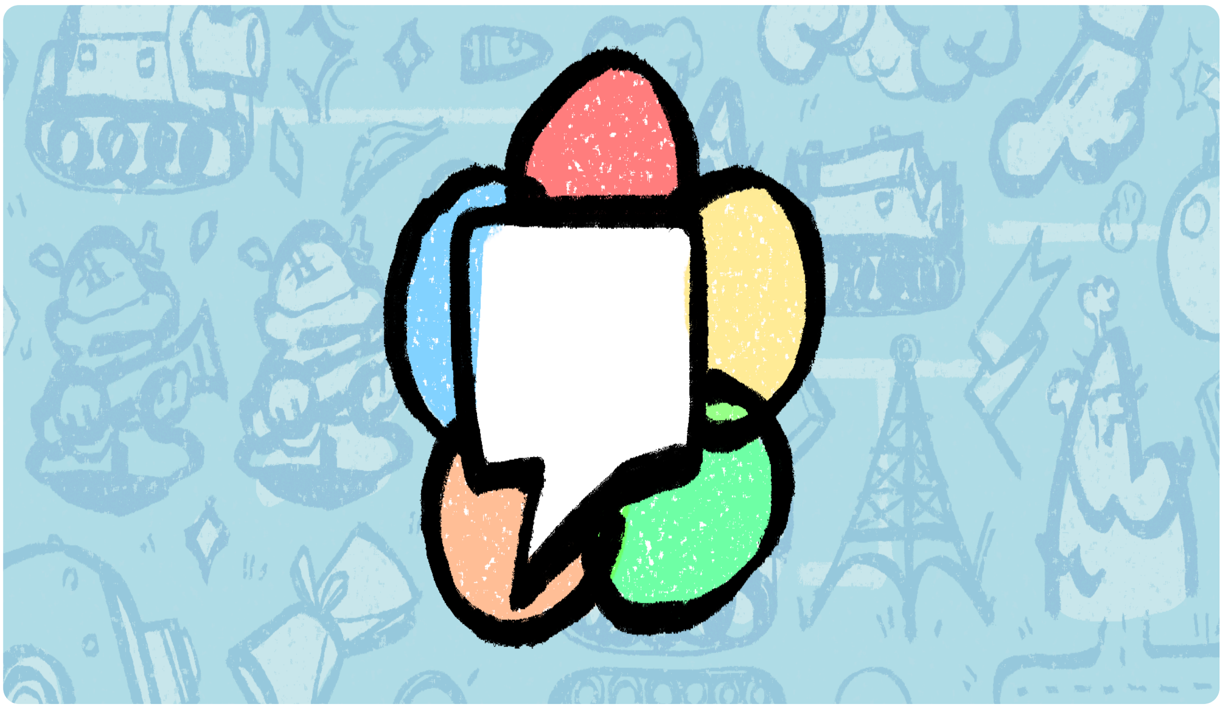


ENGINEERING & DEVELOPERS

HOW DISCORD HANDLES TWO AND HALF MILLION CONCURRENT VOICE USERS USING WEBRTC

From the very start, we made very conscious engineering and product

Jozsef Vass
September 10, 2018

decisions to keep Discord well suited for voice chat while playing your favorite game with your friends. These decisions enabled us to massively scale our operation with a small team and limited resources.

This post gives a brief overview of the different technologies Discord uses to make audio/video communications a seamless reality.

For clarity, we will use the term “guild” to represent a collection of users and channels—they are called “servers” in the client. The term “server” will instead be used here to describe our backend infrastructure.

Guiding Principles

Every audio/video communication in Discord is multiparty. Supporting large group channels (we have seen 1000 people taking turns speaking) requires client-server networking architecture because peer-to-peer networking becomes prohibitively expensive as the number of participants increases.

Routing all your network traffic through Discord servers also ensures that your IP address is never leaked whether you use text, voice, or video—preventing anyone from finding out your IP address and launching a DDoS attack against you. Routing audio/video through media servers offers other advantages as well, such as moderation. For example, administrators can disable audio/video for offending participants.

Client Architecture

Discord runs on lots of platforms.

- Web (Chrome/Firefox/Edge, etc.)
- Standalone app (Windows, MacOS, Linux)
- Phone (iOS/Android).

The only way our team can support all these platforms is to take advantage of code re-use and WebRTC. WebRTC is a specification for real-time communication comprised of networking, audio, and video components standardized by both World Wide Web Consortium and Internet Engineering Task Force. WebRTC is available in all modern browsers and also as a native library to embed into applications.

Discord's audio and video features are implemented using WebRTC. This means our browser app relies on the WebRTC implementation offered by the browser. Our desktop, iOS, and Android applications, however, make use of a single C++ media engine built on top of the WebRTC native library—specifically tailored to the needs of our users. This means that certain features work better in the installed application than in the browser. For example, in our native apps we can:

- Circumvent auto-ducking behavior of the default communications device on Windows. Ducking, or volume attenuation, means that Windows automatically reduces volume of all applications when communications device is used. This is undesirable when you are playing a game and using Discord to coordinate a raid.
- Implement our own volume control to avoid changing your global operating system volume.
- Access raw audio data to perform voice activity detection and share both game audio and video.
- Reduce your bandwidth and CPU consumption during periods of silence—even very large voice

channels only have a few concurrent speakers at any given time.

- Provide system-wide push to talk functionality.
- Send extra information along with audio/video packets (such as indicating priority speaker).

Having a customized version of WebRTC means that we need to keep up-to-date with frequent updates which is a painstaking process that we are working on automating. However, providing specific features to our gamers is well worth the extra effort.

In Discord, voice/video communication is initiated by entering a voice channel or call. This means that the communication is always initiated by the client, which reduces both client and backend complexity and also increases resilience against errors. In case of infrastructure failure, participants can simply re-connect to a new backend server.

Making It Our Own

Since we have control of the native library, we do some things differently in the native app than what you see in the browser's WebRTC implementation.

First, WebRTC relies on the Session Description Protocol (SDP) to negotiate audio/video information between participants (which can be close to ten kilobytes in size round-trip). Using the WebRTC native library allows us to use a lower level API from WebRTC (webrtc::Call) to create both send stream and receive stream. We exchange a minimal amount of information when joining a voice channel. This includes the voice backend server address and port, encryption method and keys, codec, and stream identification (about 1000 bytes).

```
webrtc::AudioSendStream* createAudioSendStream(
  uint32_t ssrc,
  uint8_t payloadType,
  webrtc::Transport* transport,
  rtc::scoped_refptr<webrtc::AudioEncoderFactor
  webrtc::Call* call)
{
    webrtc::AudioSendStream::Config config{tran
    config.rtp.ssrc = ssrc;
    config.rtp.extensions = {{"urn:ietf:params:
    config.encoder_factory = audioEncoderFactor
    const webrtc::SdpAudioFormat kOpusFormat =
    config.send_codec_spec =
      webrtc::AudioSendStream::Config::SendCode
    webrtc::AudioSendStream* audioStream = call-
    audioStream->Start();
    return audioStream;
}
```

user.cpp hosted with ❤ by GitHub view raw

Moreover, WebRTC uses Interactive Connectivity Establishment (ICE) to determine the best communication path between participants. Since every client connects to our media relay server, we do not need ICE. This allows us to provide a much more reliable connection when you're behind a NAT, as well as keep your IP address secret from other parties in the channel. Clients send periodic ping messages to ensure that the firewall remains open all the time.

Lastly, WebRTC uses the Secure Real-time Transport Protocol (SRTP) for media encryption. The encryption keys are set up using Datagram Transport Layer Security (DTLS), which is based on the Transport Layer Security protocol used in your browser every day. The native WebRTC library lets you implement your own transport layer using the webrtc::TransportAPI.

Instead of DTLS/SRTP, we decided to use the faster Salsa20 encryption. In addition, we avoid sending audio data during periods of silence—a frequent occurrence especially with larger groups. This results in significant bandwidth and CPU savings—however, both client and server must be prepared to cease receiving audio data any time and rewrite audio/video packet sequence numbers.

Because our browser app uses the browser WebRTC API, we make use of SDP/ICE/DTLS/SRTP. We exchange all necessary information between client and server (less than 1200 bytes round trip) and SDP is synthesized from this information at the clients. Our voice backend infrastructure is responsible for bridging the differences between desktop and browser applications.

Backend Architecture

There are several backend services that make voice possible, but we will focus on three of them—Discord Gateway, Discord Guilds and Discord Voice. All of our signaling servers are written in Elixir allowing lots of code re-use.

When you are online, your client maintains a WebSocket connection to a Discord Gateway (we call this the *gateway* WebSocket connection). Your client receives events through this gateway connection related to guilds, channels, messages, presence, etc.

When you are connected to a voice channel, the connection status is represented by the voice state object. The client updates its voice state object using the gateway WebSocket connection.

```elixir
defmodule VoiceStates.VoiceState do
  @type t :: %{
          session_id: String.t(),
          user_id: Number.t(),
          channel_id: Number.t() | nil,
          token: String.t() | nil,
          mute: boolean,
          deaf: boolean,
          self_mute: boolean,
          self_deaf: boolean,
          self_video: boolean,
          suppress: boolean
        }

  defstruct session_id: nil,
            user_id: nil,
            token: nil,
            channel_id: nil,
            mute: false,
            deaf: false,
            self_mute: false,
            self_deaf: false,
            self_video: false,
            suppress: false
end
```

voice_state.ex hosted with ❤ by GitHub view raw

When you join a voice channel, you are assigned to a Discord Voice server. The Discord Voice server is responsible for transmitting every member's audio to the channel. All the voice channels within a guild are assigned to the same Discord Voice server. If you are the first voice participant in the guild, Discord Guilds server is responsible for assigning a Discord Voice server to the guild using the process described below.

Assigning a Voice Server

Each voice server periodically reports its health and load, and this information is curated and placed into our service discovery system (we use etcd) as we've discussed in a previous blog post.

The Discord Guilds server watches the service discovery system and assigns the least utilized voice server in the given region to the guild. When a Discord Voice server is selected, all the voice state objects (also maintained by Discord Guilds) are pushed to voice server so it knows how to set up audio/video forwarding. Clients are also notified about the selected Discord Voice server. The client opens a *second* WebSocket connection to the voice server (we call this the *voice* WebSocket connection) which is used for setting up media forwarding and speaking indication.

When a client displays "Awaiting Endpoint," it means that the Discord Guilds server is looking for the best Discord Voice server. When a client displays “Voice Connected,” it means that the client successfully exchanged UDP messages with the selected Discord Voice server.

Discord Voice server contains two components: a signaling component and a media relay component called the selective forwarding unit or SFU. The signaling component fully controls the SFU and is responsible for generating stream identifiers and encryption keys, forwarding speaking indication, etc.

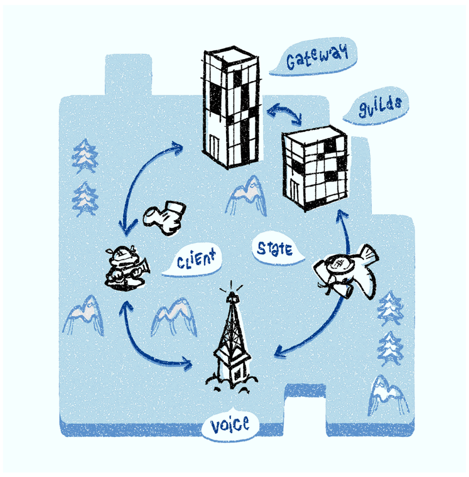
Our homegrown SFU (written in C++) is responsible for forwarding audio and video traffic within channels. Our SFU is tailored to our use case offering maximum performance and thus the lowest cost. When offending users are moderated (server mute), their audio packets are dropped. The SFU also acts as a bridge between native and browser applications. It implements a transport and encryption for both browser and standalone applications and translates between the two as it forwards media packets. Finally, the SFU is also responsible for handling the real-time control transport protocol (RTCP), which is used for video quality optimization. It collects and processes RTCP reports from receivers and notifies senders how much bandwidth is available for video.

Failover

Since it’s the only service directly accessible from the public Internet, we will focus on Discord Voice server failovers.

The signaling component constantly monitors the SFU. If the SFU crashes, it is restarted right away causing minimal service interruption (few dropped packets). The state on the SFU is reconstructed by the signaling component without any client interaction. Although SFU crashes are rare, we use the same mechanism for zero-downtime SFU updates.

When a Discord Voice server dies, it fails the periodic ping and gets removed from the service discovery system. The client also notices server failure due to the severed voice WebSocket connection and requests a *voice server ping* through the gateway WebSocket connection. The Discord Guilds server confirms the failure, consults the service discovery system, and assigns a new Discord Voice server to the guild. Discord Guilds then pushes all the voice state objects to the new voice server. Every client is notified about the new voice server and creates a voice WebSocket connection to the new voice server to start media setup.



It is quite common that a Discord Voice server suffers a DDoS attack (which we observe from the rapid increase of incoming IP packets). We perform the same procedure as for Discord Voice server failure: remove the impacted Discord Voice server from the service discovery system, select a new Discord Voice server for the guild, push all the voice state objects to the newly selected Discord Voice server, and notify clients of the new voice server for reconnection. When a DDoS attack subsides, the server is added back to the service discovery system for serving voice traffic.

When the guild owner decides to select a new voice region, we perform a very similar procedure as described above. The Discord Guilds server selects the best available Discord Voice server within the new voice region by consulting the service discovery system. It pushes all the voice state objects to the newly selected server and notifies clients about the new voice server. Clients tear down their voice WebSocket connection to the old Discord Voice server and create a new voice WebSocket connection to the new Discord Voice server.

Operating at Scale

Discord Gateway, Discord Guilds and Discord Voice are all horizontally scalable. Discord Gateway and Discord Guilds are running at Google Cloud Platform.

We are running more than 850 voice servers in 13 regions (hosted in more than 30 data centers) all over the world. This provisioning includes lots of redundancy to handle data center failures and DDoS attacks. We use a handful of providers and use physical servers in their data centers. We just recently added a South Africa region. Thanks to all our engineering efforts on both the client and server architecture, we are able to serve more than 2.6 million concurrent voice users with egress traffic of more than 220 Gbps (bits-per-second) and 120 Mpps (packets-per-second).

What's Next?

We are constantly monitoring voice quality (clients report quality metrics to our backend servers). In the future, we want to use this information to automatically detect and reduce voice degradation issues.

Although we released video chat and screen sharing about a year ago, you can currently only use it with direct messaging. When compared to audio, video requires significantly more CPU power and bandwidth. It is a challenge to balance the amount of bandwidth and CPU/GPU resources used to provide the best video quality, especially when a group of gamers in a channel may be on a range of different devices. Scalable Video Coding could be the solution to provide seamless video experience.

A big part of Discord is about sharing your gaming experiences with your friends. Screen sharing requires even more bandwidth due to higher frame rate and resolution than your ordinary webcam. We are adding hardware video encoding to our desktop application for a better experience.

We have a handful of engineers working on both client and server components and looking after operations as well. We are investing heavily in our voice and video clients and infrastructure to make Discord the best voice and video experience for gamers.

We're always looking for the next great addition to our engineering teams at Discord. If you are passionate about working with audio and video, check out our jobs page!

THE AUTHO

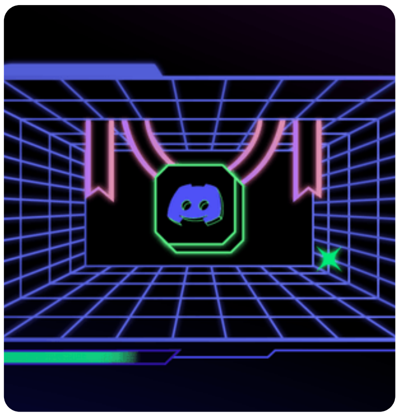
Jozsef Vass

Jozsef is a staff software engineer at Discord with more than twenty years of experience in developing scalable audio and video communications systems.

MORE FROM ENGINEERING & DEVELOPERS

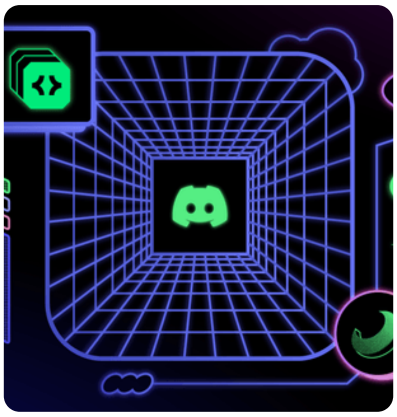
ENGINEERING & DEVELOPERS

Access: A New Portal for Managing Internal Authorization



ENGINEERING & DEVELOPERS

It's time to Pitch Your Discord Apps!!



ENGINEERING & DEVELOPERS

Headed to GDC 2024? Check Out the Latest News for Discord Game Devs

English (UK)

Product	Company	Resources	Policies
Download	About	College	Terms
Nitro	Jobs	Support	Privacy
Status	Brand	Safety	Cookie Settings
App Directory	Newsroom	Blog	Guidelines
		Feedback	Acknowledgement
		StreamKit	