

**WebRTC  
GUIDEBOOK**



# **VIDEO CALL QUALITY**

Video calls are one of the most demanding and complicated jobs that we ask our computers, tablets, and phones to do.

A video call involves encoding video and audio in real time, transmitting the encoded streams across the network and adapting quickly to changing network conditions, and finally decoding and playing the video and audio.

Humans can perceive small variations in resolution, frame rate, audio quality, and latency. So all of these operations need to happen quickly.

Despite this complexity, billions of people have devices on their desks and in their pockets that are capable of high-quality video call experiences. This guide explains what goes into delivering great video call experiences, anywhere in the world, on any device.

# Delivering high-quality and reliable video calls requires:

1. Adjusting dynamically to network conditions
2. Managing CPU use
3. Helping ensure users' cameras, microphones, and speakers are working as expected

It is important for developers who implement video calls to understand all three of these potential problem areas at a high level.

Additionally, developers will either need to be able to test across a wide variety of real-world networks and devices, or to build on top of a platform that provides full network, CPU, and device-management support.

This document will cover the following concepts:

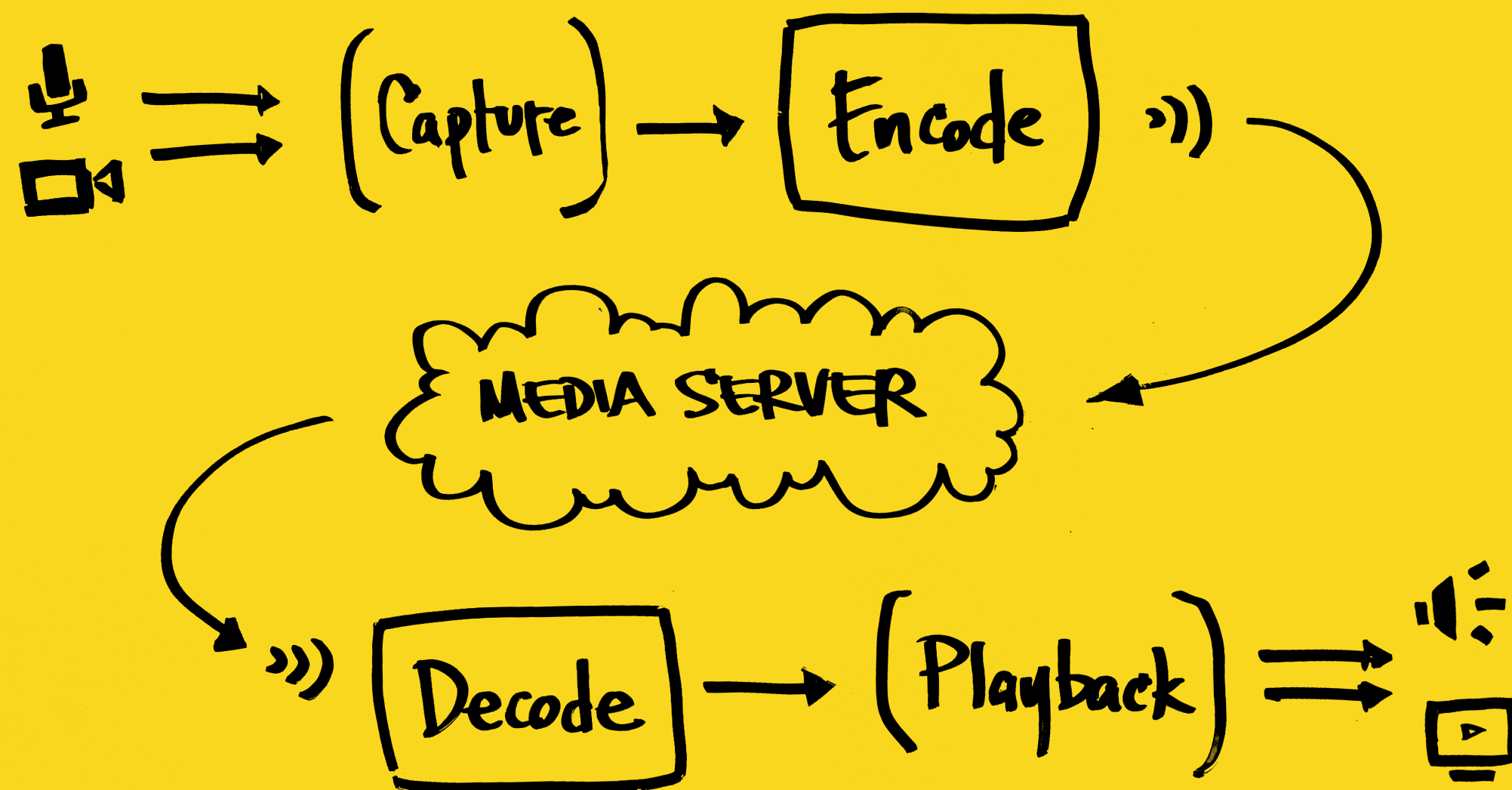
- Encoding streaming playback
- Real-time networking
- TCP, UDP, and speed testing
- Managing video call bandwidth
- Routing video
- Global infrastructure
- Firewalls
- Managing CPU use
- Managing microphones, cameras, and speakers
- Tips for professionals and power users

We also try to highlight areas where the details about video call quality can be surprising:

- Most network issues that impact video call quality happen on local WiFi, not on long-distance Internet links!
- Using the “unreliable” UDP networking protocol is better for video calls than the “reliable” TCP protocol.
- Consumer Internet speed tests are mostly not useful for testing whether a network connection will be good for video calls.
- Routing video and audio through media servers can often improve latency and lower packet loss compared to routing peer-to-peer.



## The building blocks of a video call.



Sending video and audio comprises several steps: capturing a digital data stream from a camera or microphone, processing that stream to do things like filter out background noise, then compressing the stream so that it can be sent over the Internet.

Similarly, displaying video and audio on the receiving side of a call involves assembling incoming data, decompressing the streams, possibly additional processing, then playing the video and audio in sync.

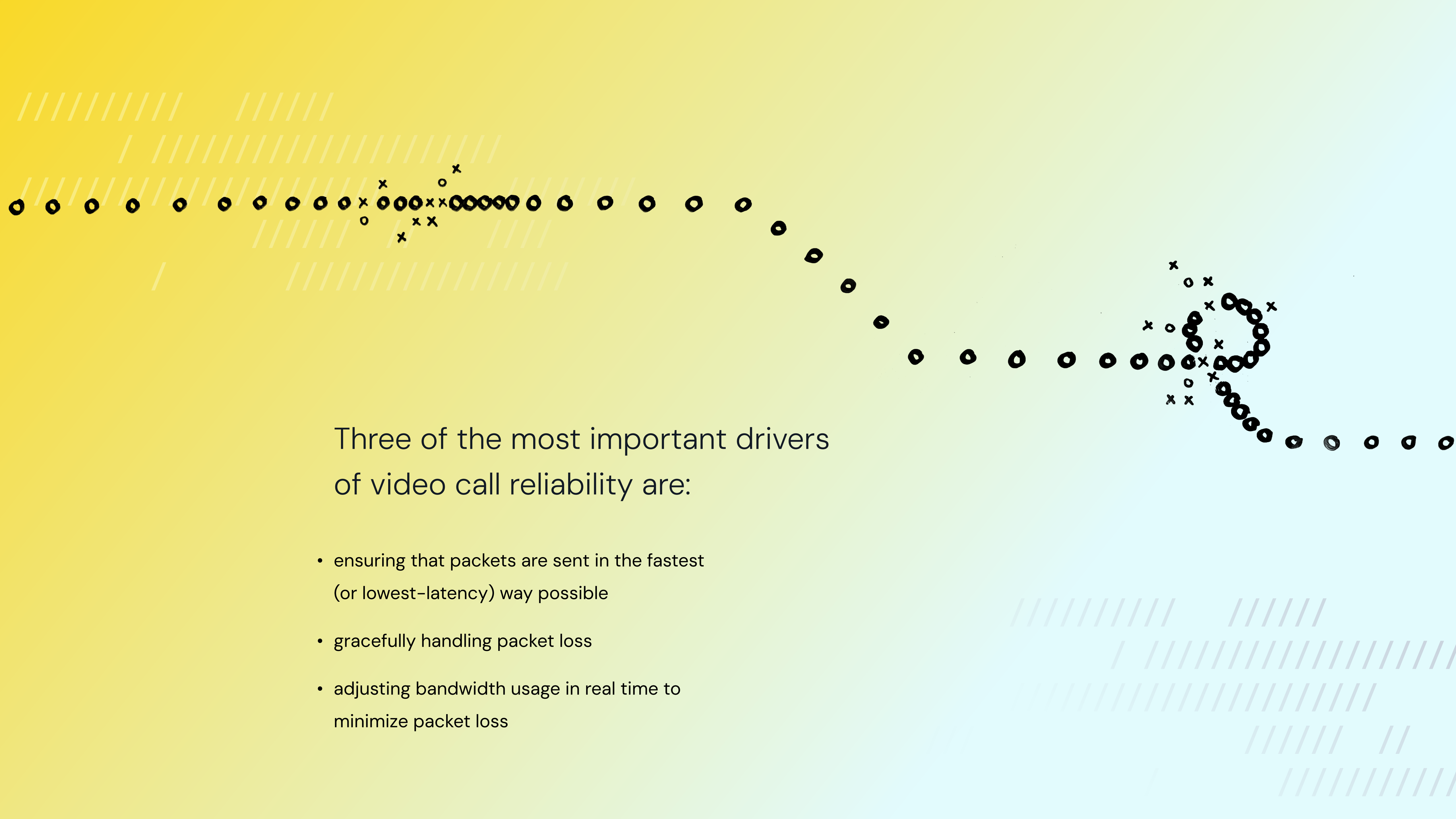
In between, the data traverses the Internet. Most video calls route data through a media server. A media server helps the devices participating in the call to adapt to changing network conditions, and can also perform additional functions like recording the call or adding real-time transcription.

Video calls are “real-time” applications. For a conversation to feel natural, the delay between a person talking and another person hearing and seeing that speech must be less than about 200ms, or one-fifth of a second.

Data streams sent over the Internet are broken up into tiny, discrete packets. Packets travel at close to the speed of light and every packet travels independently.

Overall, the Internet is very reliable. But any individual packet might be lost or delayed. Network engineers refer to packets that never arrive at their destination, or that arrive too late to be useful, as “lost” packets.

Data packets can also be sent across the Internet in different ways. For example, a packet might be routed through a Virtual Private Network server, or might be sent more directly. Some routes will be faster than others. Network engineers refer to the time it takes a packet to be sent over the network as “latency.”



Three of the most important drivers  
of video call reliability are:

- ensuring that packets are sent in the fastest  
(or lowest-latency) way possible
- gracefully handling packet loss
- adjusting bandwidth usage in real time to  
minimize packet loss

# Two low-level networking protocols carry the vast majority of Internet traffic: TCP and UDP.

UDP (user datagram protocol) is best for applications like real-time video, where latency is important. TCP (transmission control protocol) is best for applications like web pages and non-real-time video, where receiving every packet is more important than latency.

Network engineers refer to UDP as an “unreliable” or “not guaranteed to be reliable” protocol, because lost packets are just ignored at the UDP level, rather than re-transmitted. Perhaps counter-intuitively, using the “unreliable” UDP protocol is actually better for video call quality than the “reliable” TCP protocol. The real-time nature of video calls means that re-transmitting every lost packet wastes significant resources. It’s better to work around packet loss in more creative ways.



Two things are important to understand about UDP and TCP as they relate to real-time video call quality.

First, video calls should use UDP whenever possible. In some corporate environments this will require configuring firewalls to allow UDP traffic or setting up VPN services so that video traffic is not routed over the VPN. (See Firewalls on page 16)

Second, most Internet speed test sites test TCP transmission, not UDP. The results of a TCP speed test are often misleading, because a TCP test measures overall network throughput but ignores latency and packet loss. If you are testing how a network will perform for video calls, it's important to measure latency and packet loss.

# Video uses a lot of bandwidth.

The single most important component of delivering reliable, high-quality video calls is bandwidth management. Most video call issues are network issues, and most network issues can be avoided by proactively monitoring network conditions and striking the right balance between video quality and bandwidth usage.

A useful rule of thumb is that good video and audio quality can be delivered if a user's network connection can sustain about 800 kilobits per second of real-time throughput. However, users expect their video calls to work even if their network connections are not ideal. And conversely, if more bandwidth is available, it can be important for some use cases to deliver higher-quality video and audio whenever possible.

In addition, network conditions often change during a call. A nearby device downloading a big file, or even someone turning on a microwave oven, can radically reduce available bandwidth with no warning.

So a video call implementation needs to monitor available bandwidth in real time and adjust how the video is encoded and sent based on current network conditions.

Today's best video implementations manage bandwidth by combining three approaches: variable bitrate, track subscription, and simulcast.

<b>12 kb/s</b>	lowest acceptable audio quality
<b>20 kb/s</b>	good audio quality for speech
<b>160 kb/s</b>	good audio quality for music or broadcast
<b>80 kb/s</b>	lowest acceptable video quality
<b>600 kb/s</b>	good video quality for general use
<b>1,500 kb/s</b>	excellent quality for typical camera video
<b>3,000 kb/s</b>	excellent quality for high-framerate screen sharing

Variable bitrate means that the video and audio encoders used are capable of changing almost instantaneously the compression and quality targets for outgoing video and audio streams.

Track subscription means that video streams that are not currently displayed will not be sent across the network. For example, most general-purpose video call user interfaces implement a paginated or active speaker layout for large sessions. These UX approaches help to limit the number of video streams that need to be received simultaneously.

Simulcast means sending multiple video “layers” with different quality targets at the same time. For example, simulcast allows a lower-resolution stream to be received for each of the small video tiles in an active speaker layout. An efficient, configurable simulcast implementation is critical to providing high-quality, large-scale video call experiences.

There are three main ways to send video and audio packets between participants in a video call: data can be routed directly peer-to-peer, or through a media server, or through multiple media servers.

Peer-to-peer routing is simpler, requires less infrastructure, and can often be less expensive than routing through a media server. In a peer-to-peer call, each data packet is sent as directly as possible from the sender to the receiver of the audio and video.

However, peer-to-peer routing doesn't work well for calls with more than a handful of participants.

Media servers allow video calls to scale up to large sizes. In a call that is managed by a media server, each participant sends video and audio to the server. The server then forwards video and audio as needed to each client.

***Note that routing through a media server can improve quality and reliability even for 1:1 video calls.***

Routing through a server can reduce issues for clients behind firewalls.

Today's global Internet architecture has evolved to be dependent on large "backbone" connections. Effective placement of media servers near Internet backbone routes can result in lower average latencies and packet losses than is possible with peer-to-peer connections.



Routing through multiple media servers is an extension of routing through a single media server. This is called “mesh” or “cascading” routing, and allows for larger sessions and better video quality when users join a session from locations that are geographically far apart. More on mesh routing on the next page.

There is one argument for using peer-to-peer routing in regulated or extremely privacy-conscious environments: end-to-end encryption. Web browsers implement end-to-end encryption for peer-to-peer calls internally, below the level of javascript application code. If true, auditable end-to-end encryption is an application requirement, implementing video calls in a web browser and using only peer-to-peer routing can be a good approach.

## Peer-to-peer

- suitable for 1:1 calls that require very low infrastructure cost
- can be appropriate for use cases where end-to-end encryption is a requirement

## Media server

- required for large video calls
- generally better performance and reliability than peer-to-peer routing
- media server mesh:
  - supports larger numbers of participants (up to hundreds of thousands)
  - better quality video for widely dispersed users

# The ability to send data packets anywhere in the world, almost instantly, is a modern marvel.

But video calls highlight how important the “almost” is in “almost” instantly. For best call quality, it's important to situate media servers as close as possible to the users.

It takes about 80ms to send data packets between New York and London. This is comfortably within our 200ms rule of thumb for the maximum end-to-end video and audio latency that still allows comfortable, real-time conversation.

Now imagine that the data packets need to be routed through a media server, which will usually be the case for today's video call use cases.

The location of that media server matters a lot. For example, if the media server is in San Francisco, the transit time for all of the packets will approximately double. Our network latency is now 150ms. That's starting to leave very little time for processing, encoding, decoding, and playing the audio and video on each end of the connection. (Remember that end-to-end latency includes *\*all\** processing, not just network latency.)

On the other hand, if the media server is in New York or London, the short "hop" to and from the media server will take only a few milliseconds.

There is an additional benefit to global infrastructure combined with a cutting-edge mesh routing implementation. It's possible to connect each user to a media server very close to them, and route video and audio across Internet backbone connections between the media servers. This approach lowers "first hop" latencies, average transit times, and packet loss for every participant in a call.



Enterprise firewalls, Virtual Private Networks, and browser extensions can all block video call traffic.

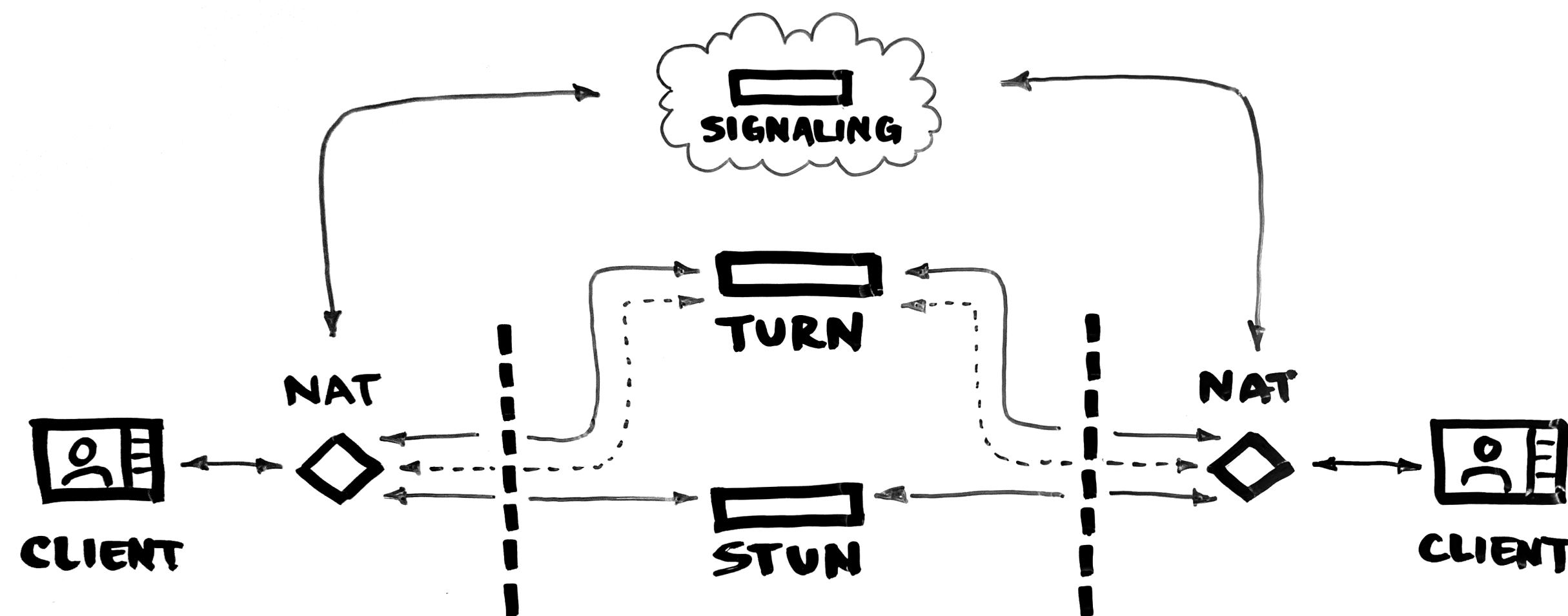
It's possible to automatically route around some traffic restrictions. In other cases, manual configuration by users or IT staff will be necessary.

In a corporate or educational setting, users may find themselves behind firewalls that block some kinds of Internet traffic. Usually, traffic is blocked in an attempt to keep systems secure, prevent exfiltration of data, or both. It is unusual today for video traffic to be entirely blocked. But even if video traffic is not entirely blocked, firewall and network configuration can still create sub-optimal routing for video traffic, which impacts call quality.

If firewalls block the UDP protocol and the data ports used by real-time video traffic, a good video call implementation will fall back to using the less efficient TCP protocol and “tunneling” video through connections that are similar to typical web-browsing data routes. The technical name for this approach is TURN (Traversal Using Relays around NAT). TURN is quite effective at allowing calls to connect. But video quality and latency will be poor.



Some firewalls will completely block traffic to and from any servers that are not pre-approved by corporate IT policy. In this case, video calls will not work at all.



A good video call implementation will detect TURN and other firewall blockage and inform the user (or the application administrator). In both cases, the best next step is to work with IT staff to change the firewall configuration to allow video traffic.

Individual users will sometimes also install firewall software, Virtual Private Network software, or browser extensions that block or degrade video traffic. Here again, a good implementation will automatically detect common problems and provide information to the user or the app administrator.

However, some software issues can be difficult to automatically diagnose. When helping a user with a video call issue, a good tech support best practice is to do a test call with all firewall software, VPN software, and browser extensions disabled.

The most common reason for out of control CPU usage in a video call is receiving too many high-resolution video streams at once.

Encoding and decoding video and audio uses a relatively large amount of CPU (and GPU) resources. And because encoding and decoding have to happen in real time, heavy contention for CPU resources will cause call quality issues.

Most general-purpose video call user interfaces default to an “active speaker” layout so that there is only one high-resolution video tile being decoded and displayed at a time. (It takes about ten times as much CPU to decode and display a 1920x1080 video stream as it does to decode and display a 320x170 video stream.)

Screen sharing presents particularly tricky challenges. Use cases for screen sharing range from viewing static, text-heavy documents that need high resolutions to be legible to sharing high-framerate video. Combining high resolutions and high framerates rapidly exhausts CPU budgets. Most video call applications default to low framerates for screen sharing, because sharing documents is more common than sharing video streams.

Especially when running inside a web browser, application-level CPU usage can easily be high enough to cause dropped video and audio frames. Usually, this will look to users like a slightly choppy or laggy video. But in the worst cases, high CPU usage causes distracting audio sync or audio quality issues.

*If video is embedded in an application that has animations, or has other features such as chat, it's important to profile CPU usage of the application as a whole.*

### Best practices for CPU usage:

- default to a “medium” video resolution for sending camera streams
- default to low framerates for screen shares, providing an option to users to increase the framerate if necessary for your use case
- select lower resolution simulcast layers on the receive side wherever possible
- design user interfaces so that fewer streams are displayed at once, especially on mobile devices
- monitor CPU usage and adjust video resolution and layout in response to high CPU usage
- profile the non-video components of the application to reduce overall CPU use
- for some use cases, consider using the H.264 video codec

Video and audio encoding and decoding technology continue to improve. Currently, the best codecs for general real-time use are VP8 for video and OPUS for audio. Today, most video call applications use these two codecs.

The H.264 video codec has excellent hardware support on many mobile devices, which can reduce CPU usage, improve performance, and lengthen battery life. But H.264 hardware support is usually limited to encoding one video stream and decoding one video stream at a time. If you know that all of your users are on mobile devices and only doing 1:1 calls, it is worth considering using H.264 instead of VP8.

**A note on video resolution.** It's very easy to see the difference between low resolutions and high (High Definition) resolutions when video is produced by professional-quality equipment and techniques. But the difference between full HD resolution and lower resolutions is usually not easy to perceive in a video call context.

The cameras that most people use for video calls have small lenses and small image sensors. They generally can't capture video at full HD resolution and quality in natural lighting situations. The best way to improve how video looks in a call is to use an inexpensive ring light or video light! Adding lighting allows inexpensive cameras to function much closer to ideal resolutions and framerates.

Additionally, the network is usually the limiting factor for real-time video. Trying to send a 1920x1080 full-resolution HD stream over a typical home WiFi network often doesn't increase the actual visual resolution compared to the same stream compressed to 1280x720 resolution.



Camera, microphone, and speaker device management cause a long tail of video call user experience issues.



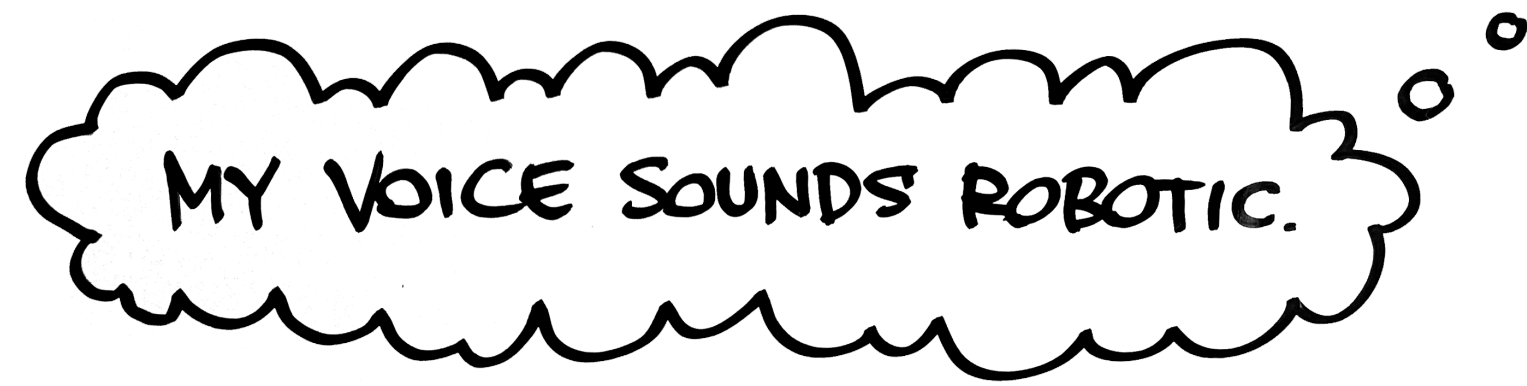
MY CAMERA ISN'T WORKING



NOBODY CAN HEAR ME.



WRONG SPEAKERS



Device issues can be particularly challenging to debug because error information is hard to generate and is usually platform-specific. For example, when running inside a web browser, camera and microphone permissions function differently than for a native app.

Another challenge is that operating system and web browser updates often introduce device-related bugs. Both iOS and macOS, for example, have a history of “two steps forward, one step back” bug fixes and regressions relating to bluetooth audio.

These issues can be difficult to debug. For example, the root cause of “nobody can hear me” could be:

- The wrong microphone is selected at the Operating System or application level
- The microphone has a hardware mute button that is activated
- The microphone is muted at the application level
- Batteries are dead on a bluetooth device
- This is a 1:1 call and the \*other\* person on the call has their speaker volume turned down or the wrong speakers selected

A good video call implementation attempts to capture as much state information as possible about hardware device activity, then translate that low-level information into simple UX that is helpful to the user.

For example, displaying microphone input audio levels in real-time in a settings dialog helps users to quickly debug whether their microphone devices are working properly.

As with network and CPU issues, testing on a large variety of real-world devices is critical. There's no other way to write code that handles long tail failure modes gracefully.

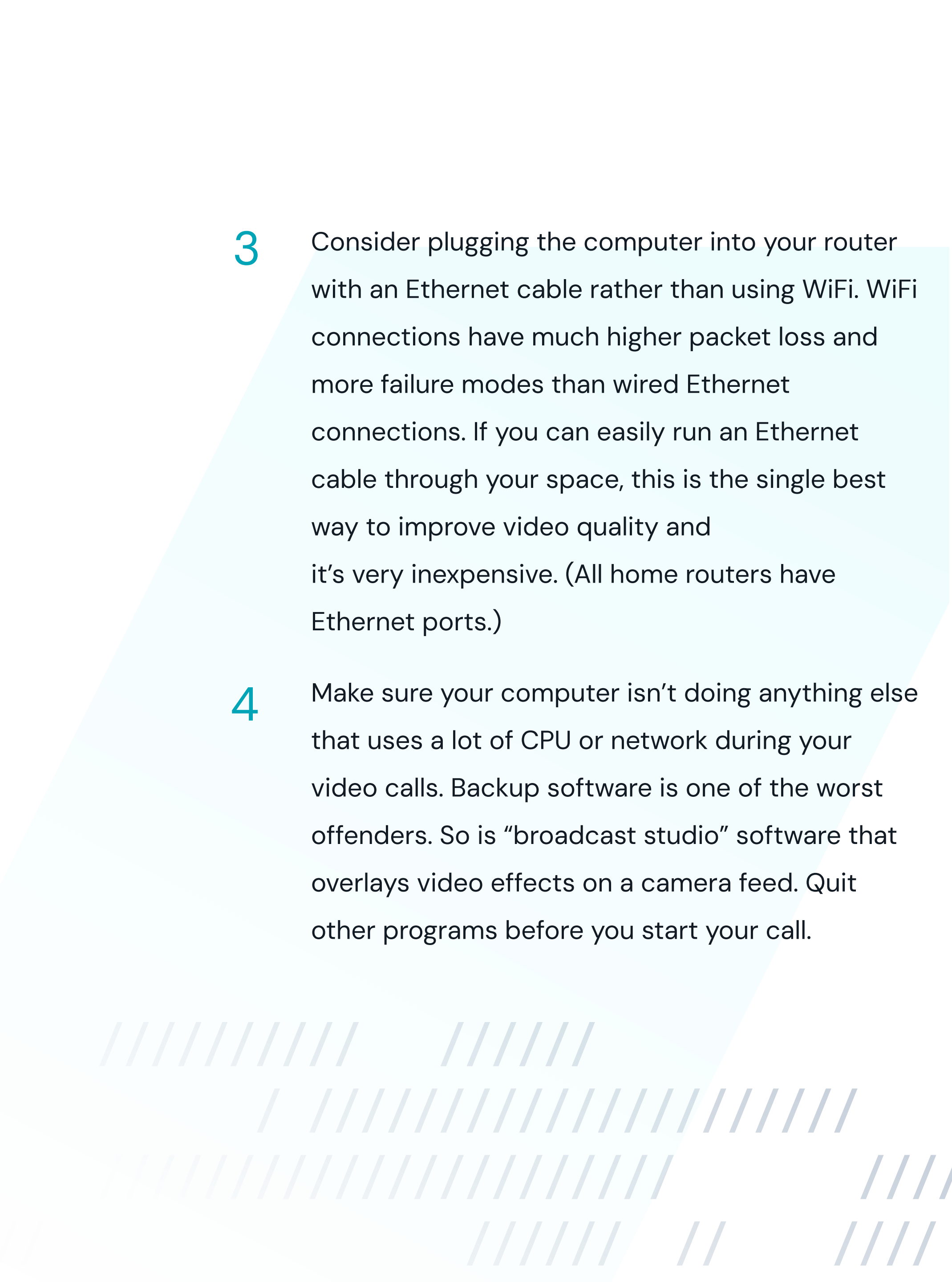

Most video call applications and platforms strive to provide a reliable call experience on any device, without asking users to change anything about their everyday behavior.

This is an important goal to aim for, and mostly achievable. However, if you support users who make money from video calls (or live streams), it's worth suggesting they think about some simple best practices for ensuring call quality and reliability.

## Tips for professional and power users:

- 1 Use a computer, not a phone or tablet. Computers have fewer thermal and battery limitations and are better suited for video calls than all but the most expensive mobile devices.
- 2 Use a relatively new computer with an up-to-date operating system. You don't have to use an expensive computer, but older hardware has more limitations and less software support than newer devices.



- 
- 
- 3 Consider plugging the computer into your router with an Ethernet cable rather than using WiFi. WiFi connections have much higher packet loss and more failure modes than wired Ethernet connections. If you can easily run an Ethernet cable through your space, this is the single best way to improve video quality and it's very inexpensive. (All home routers have Ethernet ports.)
  - 4 Make sure your computer isn't doing anything else that uses a lot of CPU or network during your video calls. Backup software is one of the worst offenders. So is "broadcast studio" software that overlays video effects on a camera feed. Quit other programs before you start your call.

- 5 Restart your video call program or web browser before starting the call. It seems strange to say this today, but the age-old advice of "quit and restart" can still avoid a significant number of device and performance issues.
- 6 Better lighting is usually the easiest way to improve video call image quality. Buying an inexpensive LED video light is likely to make a much bigger impact on how you look than buying an expensive camera.
- 7 Use a wired microphone, not bluetooth, and wear headphones if possible rather than using speakers. Bluetooth causes lots of issues with video calls—wired devices are much, much more reliable. And using headphones avoids many common causes of echo and audio distortion issues.

WEBRTC—  
THE STANDARD FOR  
VIDEO CALLS

# Most video and audio call applications today make use of a technology called WebRTC.

WebRTC, sometimes in heavily modified form, is embedded in Google Meet, Microsoft Teams, FaceTime, Facebook Messenger, and Discord. Zoom is the only widely used video call application today that is not built on WebRTC.

Routing through multiple media servers is an extension of routing through a single media server.

## **WebRTC has taken over the video calling space for two reasons:**

- The WebRTC standard is both complete enough and flexible enough to serve a wide variety of use cases: 1:1 calls, group calls, live streams, and specialized applications like professional recording tools.
- Widespread support in web browsers has made “one click, no download” video calls and video experiences possible.

WebRTC provides low-level building blocks for device management, video and audio encoding and decoding, and network connectivity.

Application developers usually don't build on top of WebRTC directly. Production video applications require functionality that is not core to WebRTC, such as session setup and state management, bandwidth management, handling differences between platforms, and global media server infrastructure. Good libraries that provide higher-level abstractions and extended functionality are available. Commercial Platform-as-a-Service offerings bundle libraries, SDKs, and infrastructure together.

# WebRTC Resources

- [Get Started with WebRTC](#) from Google
- [WebRTC for the Curious](#), an open source guide
- [Tips to improve WebRTC video call browser performance](#) from Daily
- [WebRTC Google Group](#) to keep up with changes in Chrome
- [Optimize call quality in larger calls](#) from Daily
- [Scale large video calls with dynamic simulcast layers](#) from Daily
- [Dashboard settings, metrics, and logs for WebRTC](#) from Daily
- [Call quality and beyond with WebRTC logs and metrics](#) from Daily



**The best video experiences run on daily.**

Learn more at [daily.co](https://daily.co)