Описание варианта архитектуры решения для потоковой передачи удаленного рабочего стола с использованием WebRTC

(Автор: Климов И.С., 06.2022).

В рамках данного документа рассматриваются ключевые компоненты решения, которые могут быть использованы для интерактивной потоковой передачи удаленного рабочего стола (например, в рамках инфраструктуры виртуализации рабочих мест (VDI), терминальных сессий, приложений). Рассматриваемое решение может быть использовано для создания решений для потоковой передачи как рабочих столов целиком, так и отдельных приложений.

Документ предназначен для технической аудитории, в обязанности которой входит создание, масштабирование и развертывание пользовательских решений на основе потоковой передачи. В документе предполагается, что читатель знаком с основными концепциями потоковой передачи и удаленного рабочего стола, и вам не нужно быть экспертом.

Введение

Обычно сессии удаленного рабочего стола с интенсивным использованием графики не отображаются достаточно плавно и комфортно для пользователей на платформах, использующих стандартные протоколы VNC и RDP. Одним из возможных решений этой проблемы является потоковая передача контента на рабочее место с помощью технологий, лежащих в основе проектов с открытым исходным кодом WebRTC и GStreamer. Их использование дает возможность аппаратного кодирования или декодирования потоков в реальном времени.

Вот несколько примеров использования:

- Удаленная интерактивная визуализация больших наборов данных для научных отраслей
- Оптимизированная потоковая передача динамического контента на рабочее место
- Легкие и недорогие виртуальные рабочие станции для разработчиков ПО, художников, аниматоров и дизайнеров контента
- Онлайн-конфигуратор продуктов с визуализацией в режиме реального времени для розничной торговли и промышленности

Проблема

Технология потоковой передачи с открытым исходным кодом не видела значительного скачка производительности в течение нескольких лет. Такие инструменты, как Virtual Network Computing (VNC) и Remote Desktop Protocol (RDP) по-прежнему широко используются для большинства решений для удаленного рабочего стола через Интернет. К сожалению, эти протоколы плохо подходят для современных нагрузок, требуют, чтобы пользователи загружали клиент, и их трудно защитить и настроить для сетей, защищенных брандмауэрами.

Хотя такие решения, как Guacamole и noVNC , обеспечивают веб-доступ для VNC и RDP, они не соответствуют требованиям к производительности и задержке для графически интенсивных рабочих нагрузок. Для большинства этих приложений ожидается не менее 30 кадров в секунду при разрешении 1080p или лучше, чего трудно достичь без специализированного ускорения. Многие современные серверные решения имеют встроенные или устанавливаемые дополнительно графические процессоры, что дает возможность аппаратно кодировать поток в реальном времени. Это может значительно повысить производительность технологий потоковой передачи за счет снижения использования CPU и полосы пропускания, необходимых для доставки контента. К сожалению, большинство решений с открытым исходным кодом, основанных на протоколах VNC и RDP, не поддерживают аппаратное кодирование.

В следующей таблице показано сравнение некоторых популярных возможностей потоковой передачи.

Решение	**Поддержка Web**	**Аппаратное кодирование**	**Аппаратное декодирование**
VNC	Нет	Нет	Нет
RDP	Нет	Нет	Нет

Guacamole	Да	Нет	Canvas HTML5
NoVNC	Да	Нет	Canvas HTML5

Несмотря на то, что существуют коммерчески доступные инструменты удаленного рабочего стола, такие как PCoIP от Teradici или HDX от Citrix , в этом исследовании рассматривается решение с открытым исходным кодом, которое может быть разработано для практического использования.

Решение

WebRTC — это современный набор протоколов, предназначенный для безопасной потоковой передачи видео, аудио и произвольных данных с малой задержкой. Современные платформы могут декодировать видеопотоки H.264 без каких-либо дополнительных плагинов. Если у клиента есть локальный GPU с аппаратным декодированием видео, WebRTC автоматически использует GPU для ускорения декодирования потока, что снимает нагрузку с CPU и повышает частоту кадров. HTML5 Video Element — это единственный необходимый элемент разметки, который позволяет выполнять потоковую передачу контента, потоковую передачу одного приложения или варианты использования удаленного рабочего стола в полноэкранном режиме.

WebRTC включает в себя следующие ключевые функции:

- Поддержка типовых браузеров
- Форматы кодирования с аппаратным ускорением
- Встроенные возможности защиты передаваемых данных
- Peer-to-Peer коммуникация связь
- Способность корректно работать через шлюзы NAT и брандмауэры

Хотя WebRTC намного сложнее, чем решения VNC и RDP, это один из немногих доступных в настоящее время вариантов, обеспечивающих высокую частоту кадров и потоковую передачу с низкой пропускной способностью.

Примеры технологий, использующих WebRTC, включают следующее:

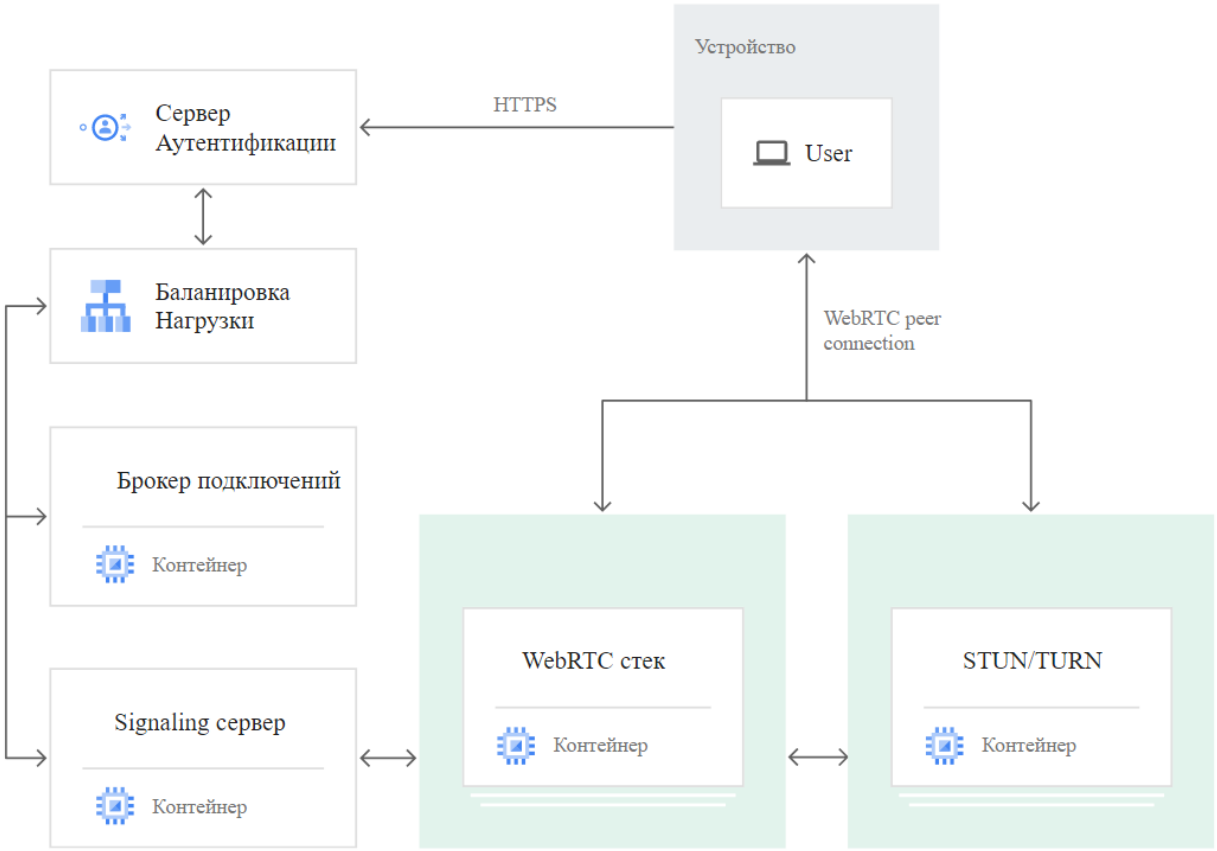
- Google Meet

- Chrome Remote Desktop
- Parsec Gaming
- Twilio
- FB Messenger

GStreamer — это инструмент с открытым исходным кодом для построения конвейеров потоковой передачи. Он поддерживает множество источников, форматов и приемников, а также включает поддержку WebRTC. В рамках исследования показано, как использовать инструмент GStreamer и аппаратный видеокодер (NVENC) для потоковой передачи ускоренных рабочих нагрузок в веб-браузер.

Описание архитектуры системы потоковой передачи

Стек потоковой передачи WebRTC общего назначения может быть развернут в контейнеризованной среде и сделан доступным для отдельных пользователей. После того, как аутентифицированный сеанс установлен, пользователи подключаются напрямую к экземпляру WebRTC, используя одноранговое соединение.



Область применения стека WebRTC

WebRTC лучше всего использовать, когда быстрый отклик и низкие задержки при работе пользователя имеют определяющее значение для приложения. Программное обеспечение для доступа к удаленным рабочим столам и приложениям является одним из наиболее требовательных к отклику и задержкам. WebRTC менее эффективен для устойчивой к задержкам потоковой передачи, такой как доставка предварительно записанного или обработанного видеоконтента.

Использование WebRTC целесообразно для любой из следующих целей:

- Доставка динамичного изображения удаленного рабочего стола малой задержкой на ПК или мобильное устройство
- Потоковая передача бинарных данных или событий с малой задержкой, таких как ввод с клавиатуры, мыши или иных периферийных устройств.

- Потоковая передача 3D-графики на ПК или мобильное устройство

Использование WebRTC как правило нецелесообразно для следующих задач:

- Крупномасштабное распространение предварительно записанных или обработанных медиафайлов.
- Потоковая передача видеоформатов, не поддерживаемых современными браузерами.
- Обеспечение совместимости со старыми браузерами.

Основные концепции

В этом разделе представлены ключевые термины и некоторые сведения о сетевом взаимодействии компонентов, установлении соединения и защите соединения с помощью DTLS.

Терминология

Поскольку рассматриваемая архитектура состоит из нескольких протоколов и технологий, решение на основе WebRTC может выглядеть довольно сложным. Однако большая часть этой сложности абстрагируется через API-интерфейсы.

Безопасность транспортного уровня дейтаграмм (DTLS)

Реализация спецификации безопасности транспортного уровня, которую можно использовать по протоколу пользовательских дейтаграмм (UDP). WebRTC требует, чтобы все данные были защищены при передаче и использует DTLS для защиты передачи данных.

Организация интерактивного подключения (ICE)

Метод, используемый WebRTC для обнаружения оптимального способа создания одноранговогo соединения. Одноранговые узлы обмениваются кандидатами ICE, которые расставляются согласно приоритету, пока не будет согласован общий метод подключения.

RTCPeerConnection

Объект JavaScript, используемый для создания соединения WebRTC. Исходный код адаптера WebRTC предоставляет стандартный интерфейс, поэтому не требуется создавать собственный код для каждой конкретной реализации клиентского окружения.

Протокол описания сеанса (SDP)

Конфигурация носителей и соединений, а также возможности, которыми обмениваются одноранговые узлы во время установления соединения.

Утилиты обхода сеанса для NAT (STUN)

Внешняя служба, используемая одноранговыми узлами для обнаружения их реального внешнего IP-адреса, если они находятся за брандмауэром или шлюзом NAT.

Сервис Signaling

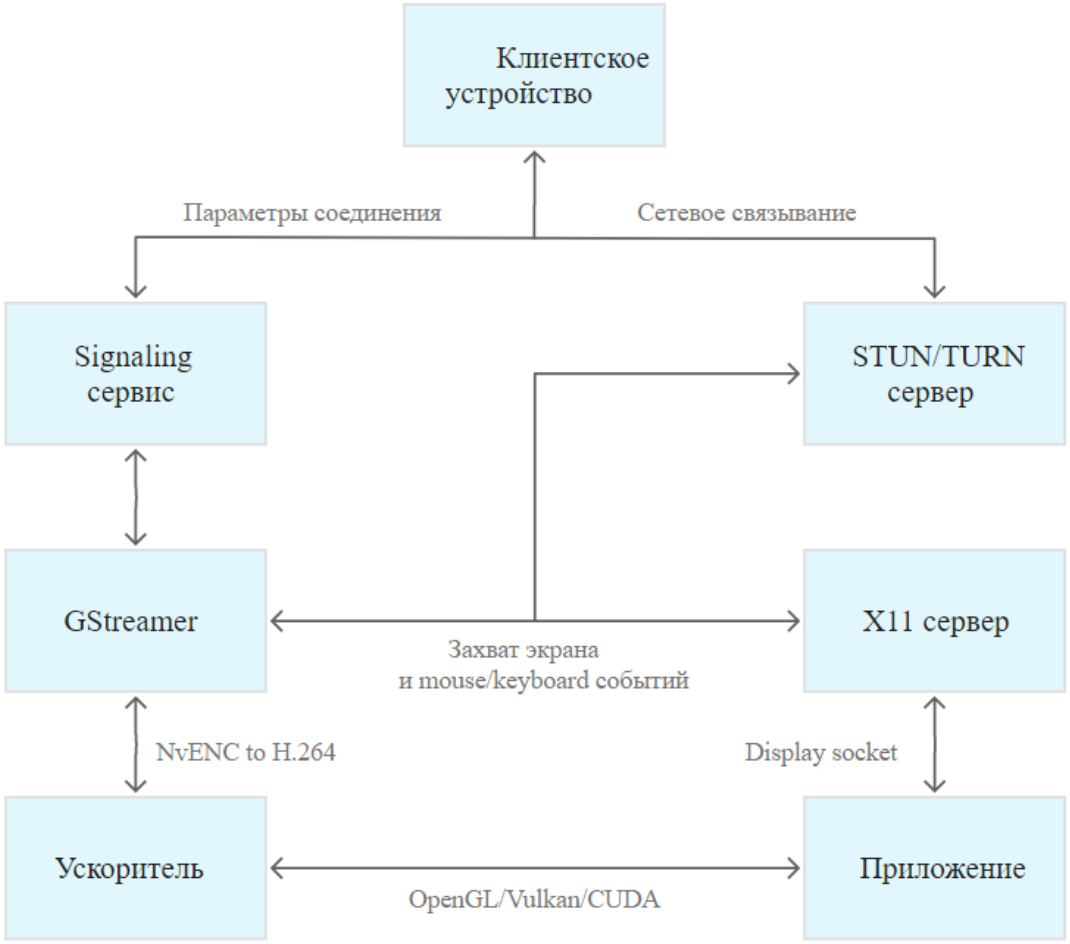
Внешний сервис, используемый одноранговыми соединениями, не включенный в спецификацию WebRTC, но необходимый для установления соединения. Хотя формальной спецификации для сигнализации не существует, обычно используется WebSocket или расширяемый протокол обмена сообщениями и присутствия (XMPP) .

Сервис для обхода NAT (TURN)

Внешняя служба, используемая одноранговыми узлами в качестве ретранслятора, если во время согласования кандидата ICE невозможно обнаружить метод прямого однорангового соединения.

Конвейер GStreamer

На следующей диаграмме GStreamer служит механизмом потоковой передачи и взаимодействует с необходимыми компонентами, такими как захват из X11, сигнализация, STUN/TURN, пользовательский ввод из веб-интерфейса и аппаратный ускоритель сжатия.



GStreamer имеет конвейерную модель обработки . Каждый элемент конвейера может действовать как источник или приемник. Для потоковой передачи дисплея X11 в браузер с помощью WebRTC требуется привязать к `webrtcbin` элементу несколько элементов :

Элементы конвейера выполняют следующие действия:

- `ximagesrc` захватывает дисплей X11 из сокета Unix со скоростью 60 кадров в секунду в буфер формата RGB.
- `videoconvert` преобразует буфер RGB в формат, совместимый с NVENC.
- `nvh264enc` кодирует буфер в H.264 на графическом процессоре, используя NVENC.

- `rtph264pay` создает полезную нагрузку пакета RTP, отправляемую через одноранговое соединение.
- `webrtcbin` обрабатывает установление соединения WebRTC и согласование параметров подключения.

Пример конвейера GStreamer

В следующих разделах подробно описаны основные элементы конвейера, их функции и способы настройки. Вы можете изменить фактические значения в соответствии с рабочей нагрузкой потоковой передачи.

В следующих примерах кода показан полный конвейер, используемый в этом примере:

gst-webrtc-app/gstwebrtc_app.py

```
def build_webrtcbin_pipeline(self):
    """Adds the webrtcbin elments to the pipeline.

    The video and audio pipelines are linked to this in the
        build_video_pipline() and build_audio_pipeline() methods.
    """

    # Create webrtcbin element named app
    self.webrtcbin = Gst.ElementFactory.make("webrtcbin", "app")

    # The bundle policy affects how the SDP is generated.
    # This will ultimately determine how many tracks the browser receives.
    # Setting this to max-compat will generate separate tracks for
    # audio and video.
    # See also: https://webrtcstandards.info/sdp-bundle/
    self.webrtcbin.set_property("bundle-policy", "max-compat")

    # Connect signal handlers
    self.webrtcbin.connect(
        'on-negotiation-needed', lambda webrtcbin:
self.__on_negotiation_needed(webrtcbin))
    self.webrtcbin.connect('on-ice-candidate', lambda webrtcbin,
mlineindex,
                           candidate: self.__send_ice(webrtcbin,
mlineindex, candidate))

    # Add STUN server
```

```
    if self.stun_server:
        self.webrtcbin.set_property("stun-server", self.stun_server)

    # Add TURN server
    if self.turn_server:
        logger.info("adding TURN server: %s" % self.turn_server)
        self.webrtcbin.emit("add-turn-server", self.turn_server)

    # Add element to the pipeline.
    self.pipeline.add(self.webrtcbin)
```

gst-webrtc-app/gstwebrtc_app.py

```
def build_video_pipeline(self):
    """Adds the RTP video stream to the pipeline.
    """

    # Create ximagesrc element named x11
    # Note that when using the ximagesrc plugin, ensure that the X11
server was
    # started with shared memory support: '+extension MIT-SHM' to achieve
    # full frame rates.
    # You can check if XSHM is in use with the following command:
    #   GST_DEBUG=default:5 gst-launch-1.0 ximagesrc ! fakesink
num-buffers=1 2>&1 |grep -i xshm
    ximagesrc = Gst.ElementFactory.make("ximagesrc", "x11")

    # disables display of the pointer using the XFixes extension,
    # common when building a remote desktop interface as the clients
    # mouse pointer can be used to give the user perceived lower latency.
    # This can be programmatically toggled after the pipeline is started
    # for example if the user is viewing full screen in the browser,
    # they may want to revert to seeing the remote cursor when the
    # client side cursor disappears.
    ximagesrc.set_property("show-pointer", 0)

    # Tells GStreamer that you are using an X11 window manager or
    # compositor with off-screen buffer. If you are not using a
    # window manager this can be set to 0. It's also important to
    # make sure that your X11 server is running with the XSHM extension
    # to ensure direct memory access to frames which will reduce latency.
    ximagesrc.set_property("remote", 1)

    # Defines the size in bytes to read per buffer. Increasing this from
    # the default of 4096 bytes helps performance when capturing high
    # resolutions like 1080P, and 2K.
```

```
    ximagesrc.set_property("blocksize", 16384)

    # The X11 XDamage extension allows the X server to indicate when a
    # regions of the screen has changed. While this can significantly
    # reduce CPU usage when the screen is idle, it has little effect with
    # constant motion. This can also have a negative consequences with
H.264
    # as the video stream can drop out and take several seconds to recover
    # until a valid I-Frame is received.
    # Set this to 0 for most streaming use cases.
    ximagesrc.set_property("use-damage", 0)

    # Create capabilities for videoconvert
    videoconvert_caps = Gst.caps_from_string("video/x-raw")

    # Setting the framerate=60/1 capability instructs the ximagesrc
element
    # to generate buffers at 60 frames per second (FPS).
    # The higher the FPS, the lower the latency so this parameter is one
    # way to set the overall target latency of the pipeline though keep in
    # mind that the pipeline may not always perfom at the full 60 FPS.
    videoconvert_caps.set_value("framerate", Gst.Fraction(self.framerate,
1))

    # Create a capability filter for the videoconvert_caps
    videoconvert_capsfilter = Gst.ElementFactory.make("capsfilter")
    videoconvert_capsfilter.set_property("caps", videoconvert_caps)

    # Upload buffers from ximagesrc directly to CUDA memory where
    # the colorspace conversion will be performed.
    cudaupload = Gst.ElementFactory.make("cudaupload")

    # Convert the colorspace from BGRx to NVENC compatible format.
    # This is performed with CUDA which reduces the overall CPU load
    # compared to using the software videoconvert element.
    cudaconvert = Gst.ElementFactory.make("cudaconvert")

    # Convert ximagesrc BGRx format to I420 using cudaconvert.
    # This is a more compatible format for client-side software decoders.
    cudaconvert_caps =
Gst.caps_from_string("video/x-raw(memory:CUDAMemory)")
    cudaconvert_caps.set_value("format", "I420")
    cudaconvert_capsfilter = Gst.ElementFactory.make("capsfilter")
    cudaconvert_capsfilter.set_property("caps", cudaconvert_caps)

    # Create the nvh264enc element named nvenc.
```

```
    # This is the heart of the video pipeline that converts the raw
    # frame buffers to an H.264 encoded byte-stream on the GPU.
    nvh264enc = Gst.ElementFactory.make("nvh264enc", "nvenc")

    # The initial bitrate of the encoder in bits per second.
    # Setting this to 0 will use the bitrate from the NVENC preset.
    # This parameter can be set while the pipeline is running using the
    # set_video_bitrate() method. This helps to match the available
    # bandwidth. If set too high, the cliend side jitter buffer will
    # not be unable to lock on to the stream and it will fail to render.
    nvh264enc.set_property("bitrate", 2000)

    # Rate control mode tells the encoder how to compress the frames to
    # reach the target bitrate. A Constant Bit Rate (CBR) setting is best
    # for streaming use cases as bit rate is the most important factor.
    # A Variable Bit Rate (VBR) setting tells the encoder to adjust the
    # compression level based on scene complexity, something not needed
    # when streaming in real-time.
    nvh264enc.set_property("rc-mode", "cbr")

    # Group of Pictures (GOP) size is the distance between I-Frames that
    # contain the full frame data needed to render a whole frame.
    # Infinite GOP is best for streaming because it reduces the number
    # of large I-Frames being transmitted. At higher resolutions, these
    # I-Frames can dominate the bandwidth and add additional latency.
    # With infinite GOP, you can use a higher bit rate to increase quality
    # without a linear increase in total bandwidth.
    # A negative consequence when using infinite GOP size is that
    # when packets are lost, it may take the decoder longer to recover.
    # NVENC supports infinite GOP by setting this to -1.
    nvh264enc.set_property("gop-size", -1)

    # Instructs encoder to handle Quality of Service (QOS) events from
    # the rest of the pipeline. Setting this to true increases
    # encoder stability.
    nvh264enc.set_property("qos", True)

    # The NVENC encoder supports a limited nubmer of encoding presets.
    # These presets are different than the open x264 standard.
    # The presets control the picture coding technique, bitrate,
    # and encoding quality.
    # The low-latency-hq is the NVENC preset reccomended for streaming.
    #
    # See this link for details on each preset:
```

```
    #
https://streamquality.report/docs/report.html#1080p60-nvenc-h264-picture-q
uality
    nvh264enc.set_property("preset", "low-latency-hq")

    # Set the capabilities for the nvh264enc element.
    nvh264enc_caps = Gst.caps_from_string("video/x-h264")

    # Sets the H.264 encoding profile to one compatible with WebRTC.
    # The high profile is used for streaming HD video.
    # Browsers only support specific H.264 profiles and they are
    # coded in the RTP payload type set by the rtph264pay_caps below.
    nvh264enc_caps.set_value("profile", "high")

    # Create a capability filter for the nvh264enc_caps.
    nvh264enc_capsfilter = Gst.ElementFactory.make("capsfilter")
    nvh264enc_capsfilter.set_property("caps", nvh264enc_caps)

    # Create the rtph264pay element to convert buffers into
    # RTP packets that are sent over the connection transport.
    rtph264pay = Gst.ElementFactory.make("rtph264pay")

    # Set the capabilities for the rtph264pay element.
    rtph264pay_caps = Gst.caps_from_string("application/x-rtp")

    # Set the payload type to video.
    rtph264pay_caps.set_value("media", "video")

    # Set the video encoding name to match our encoded format.
    rtph264pay_caps.set_value("encoding-name", "H264")

    # Set the payload type to one that matches the encoding profile.
    # Payload number 123 corresponds to H.264 encoding with the high
profile.
    # Other payloads can be derived using WebRTC specification:
    #   https://tools.ietf.org/html/rfc6184#section-8.2.1
    rtph264pay_caps.set_value("payload", 123)

    # Create a capability filter for the rtph264pay_caps.
    rtph264pay_capsfilter = Gst.ElementFactory.make("capsfilter")
    rtph264pay_capsfilter.set_property("caps", rtph264pay_caps)

    # Add all elements to the pipeline.
    self.pipeline.add(ximagesrc)
    self.pipeline.add(videoconvert_capsfilter)
    self.pipeline.add(cudaupload)
```

```
        self.pipeline.add(cudaconvert)
        self.pipeline.add(cudaconvert_capsfilter)
        self.pipeline.add(nvh264enc)
        self.pipeline.add(nvh264enc_capsfilter)
        self.pipeline.add(rtph264pay)
        self.pipeline.add(rtph264pay_capsfilter)

        # Link the pipeline elements and raise exception of linking fails
        # due to incompatible element pad capabilities.
        if not Gst.Element.link(ximagesrc, videoconvert_capsfilter):
            raise GSTWebRTCAppError("Failed to link ximagesrc ->
videoconvert")

        if not Gst.Element.link(videoconvert_capsfilter, cudaupload):
            raise GSTWebRTCAppError(
                "Failed to link videoconvert_capsfilter -> cudaupload")

        if not Gst.Element.link(cudaupload, cudaconvert):
            raise GSTWebRTCAppError(
                "Failed to link cudaupload -> cudaconvert")

        if not Gst.Element.link(cudaconvert, cudaconvert_capsfilter):
            raise GSTWebRTCAppError(
                "Failed to link cudaconvert -> cudaconvert_capsfilter")

        if not Gst.Element.link(cudaconvert_capsfilter, nvh264enc):
            raise GSTWebRTCAppError(
                "Failed to link cudaconvert_capsfilter -> nvh264enc")

        if not Gst.Element.link(nvh264enc, nvh264enc_capsfilter):
            raise GSTWebRTCAppError(
                "Failed to link nvh264enc -> nvh264enc_capsfilter")

        if not Gst.Element.link(nvh264enc_capsfilter, rtph264pay):
            raise GSTWebRTCAppError(
                "Failed to link nvh264enc_capsfilter -> rtph264pay")

        if not Gst.Element.link(rtph264pay, rtph264pay_capsfilter):
            raise GSTWebRTCAppError(
                "Failed to link rtph264pay -> rtph264pay_capsfilter")

        # Link the last element to the webrtcbin
        if not Gst.Element.link(rtph264pay_capsfilter, self.webrtcbin):
            raise GSTWebRTCAppError(
                "Failed to link rtph264pay_capsfilter -> webrtcbin")
```

Выбор типа полезной нагрузки H.264

Элемент `rtph264pay` инкапсулирует видеопоток в пакеты RTP. Свойства, установленные для этого элемента, должны совпадать со свойствами, сгенерированными получателем SDP.

Пример SDP и различных типов полезной нагрузки можно протестировать, посетив тестовую страницу WebRTC и запустив тест с установленным флажком «Требовать видео H.264».

Вывод SDP, отображаемый во время теста, показывает профили, которые поддерживает устройство:

```
a=rtpmap:102 H264/90000
a=fmtp:102
level-asymmetry-allowed=1;packetization-mode=1;profile-level-id=42001f
a=rtpmap:127 H264/90000
a=fmtp:127
level-asymmetry-allowed=1;packetization-mode=0;profile-level-id=42001f
a=rtpmap:125 H264/90000
a=fmtp:125
level-asymmetry-allowed=1;packetization-mode=1;profile-level-id=42e01f
a=rtpmap:108 H264/90000
a=fmtp:108
level-asymmetry-allowed=1;packetization-mode=0;profile-level-id=42e01f
a=rtpmap:124 H264/90000
a=fmtp:124
level-asymmetry-allowed=1;packetization-mode=1;profile-level-id=4d0032
a=rtpmap:123 H264/90000
a=fmtp:123
level-asymmetry-allowed=1;packetization-mode=1;profile-level-id=640032
```

Параметры в SDP помогают определить номер полезной нагрузки для использования. В спецификации также отмечается, что декодеры могут обрабатывать профили, отличные от тех, которые переданы в SDP. Например, если кодирование происходит с высоким профилем, но отправляется базовый тип полезной нагрузки (102), профиль все равно будет декодирован.

Первый шестнадцатеричный байт `profile-level-id` используется для описания профиля, который должен использовать в конвейере GStreamer. Поскольку

конвейер GStreamer устанавливает высокий профиль кодировщика , определяется payload=123 .

Тип полезной нагрузки	Идентификатор уровня профиля	Режим пакетирования	Профиль GStreamer
102	42 001f	1	исходный уровень
127	42 001f	0	исходный уровень
125	42 e01f	1	исходный уровень
108	42 e01f	0	исходный уровень
124	4д 0032	1	главный
123	64 0032	1	высокая

Аудио конвейер

Отправка аудио через WebRTC также возможна с помощью GStreamer. Этот участок конвейера подключается к локальному серверу PulseAudio через TCP, преобразует его в формат Opus и передает в виде RTP с использованием соответствующего номера полезной нагрузки SDP. Этот процесс требует, чтобы работающий сервер PulseAudio прослушивал адрес `127.0.0.1` и чтобы

приложение отправляло звук в подсистему Linux PulseAudio по умолчанию, как это делает большинство.

gst-webrtc-app/gstwebrtc_app.py

```
def build_audio_pipeline(self):
    """Adds the RTP audio stream to the pipeline.
    """

    # Create element for receiving audio from pulseaudio.
    pulsesrc = Gst.ElementFactory.make("pulsesrc", "pulsesrc")

    # Let the audio source provide the global clock.
    # This is important when trying to keep the audio and video
    # jitter buffers in sync. If there is skew between the video and audio
    # buffers, features like NetEQ will continuously increase the size of
the
    # jitter buffer to catch up and will never recover.
    pulsesrc.set_property("provide-clock", True)

    # Apply stream time to buffers, this helps with pipeline
synchronization.
    pulsesrc.set_property("do-timestamp", True)

    # Encode the raw pulseaudio stream to opus format which is the
    # default packetized streaming format for the web.
    opusenc = Gst.ElementFactory.make("opusenc", "opusenc")

    # Set audio bitrate to 64kbps.
    # This can be dynamically changed using set_audio_bitrate()
    opusenc.set_property("bitrate", 64000)

    # Create the rtpopuspay element to convert buffers into
    # RTP packets that are sent over the connection transport.
    rtpopuspay = Gst.ElementFactory.make("rtpopuspay")

    # Insert a queue for the RTP packets.
    rtpopuspay_queue = Gst.ElementFactory.make("queue",
"rtpopuspay_queue")

    # Make the queue leaky, so just drop packets if the queue is behind.
    rtpopuspay_queue.set_property("leaky", True)

    # Set the queue max time to 16ms (16000000ns)
    # If the pipeline is behind by more than 1s, the packets
    # will be dropped.
```

```
    # This helps buffer out latency in the audio source.
    rtpopuspay_queue.set_property("max-size-time", 16000000)

    # Set the other queue sizes to 0 to make it only time-based.
    rtpopuspay_queue.set_property("max-size-buffers", 0)
    rtpopuspay_queue.set_property("max-size-bytes", 0)

    # Set the capabilities for the rtpopuspay element.
    rtpopuspay_caps = Gst.caps_from_string("application/x-rtp")

    # Set the payload type to audio.
    rtpopuspay_caps.set_value("media", "audio")

    # Set the audio encoding name to match our encoded format.
    rtpopuspay_caps.set_value("encoding-name", "OPUS")

    # Set the payload type to match the encoding format.
    # A value of 96 is the default that most browsers use for Opus.
    # See the RFC for details:
    #   https://tools.ietf.org/html/rfc4566#section-6
    rtpopuspay_caps.set_value("payload", 96)

    # Create a capability filter for the rtpopuspay_caps.
    rtpopuspay_capsfilter = Gst.ElementFactory.make("capsfilter")
    rtpopuspay_capsfilter.set_property("caps", rtpopuspay_caps)

    # Add all elements to the pipeline.
    self.pipeline.add(pulsesrc)
    self.pipeline.add(opusenc)
    self.pipeline.add(rtpopuspay)
    self.pipeline.add(rtpopuspay_queue)
    self.pipeline.add(rtpopuspay_capsfilter)

    # Link the pipeline elements and raise exception of linking fails
    # due to incompatible element pad capabilities.
    if not Gst.Element.link(pulsesrc, opusenc):
        raise GSTWebRTCAppError("Failed to link pulsesrc -> opusenc")

    if not Gst.Element.link(opusenc, rtpopuspay):
        raise GSTWebRTCAppError("Failed to link opusenc -> rtpopuspay")

    if not Gst.Element.link(rtpopuspay, rtpopuspay_queue):
        raise GSTWebRTCAppError("Failed to link rtpopuspay ->
rtpopuspay_queue")

    if not Gst.Element.link(rtpopuspay_queue, rtpopuspay_capsfilter):
```

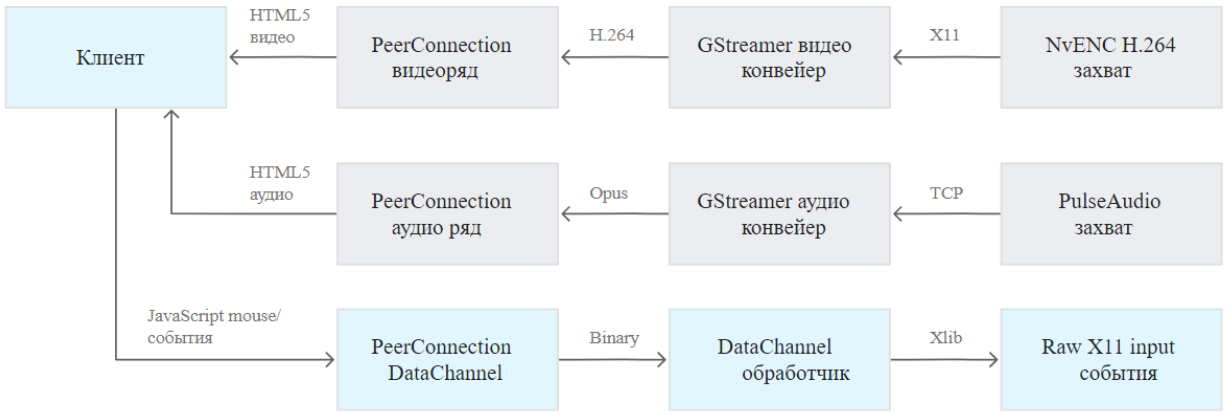
```
        raise GSTWebRTCAppError(
            "Failed to link rtpopuspay_queue -> rtpopuspay_capsfilter")

    # Link the last element to the webrtcbin
    if not Gst.Element.link(rtpopuspay_capsfilter, self.webrtcbin):
        raise GSTWebRTCAppError(
            "Failed to link rtpopuspay_capsfilter -> webrtcbin")
```

Обработка входных данных

Потоковая передача событий клавиатуры, мыши и геймпада в ваше приложение делает его интерактивным. Канал `RTCPeerConnection` данных — это произвольный поток данных, связанный с одноранговым соединением, в котором используются те же методы соединения, что и для видео- и аудиодорожек. Каналы данных используют протокол передачи управления потоком (SCTP) для передачи данных по одноранговому соединению. По этому каналу можно отправлять любые данные, а его интерфейс похож на WebSocket. Каналы данных поддерживаются в GStreamer начиная с версии 1.16 .

События клавиатуры и мыши перехватываются с помощью их собственных обработчиков событий JavaScript. Затем они преобразуются в двоичные команды и отправляются по каналу данных, где события декодируются и передаются на сервер X11. Следующая диаграмма иллюстрирует различные потоки и форматы к принимающему клиенту и отправляющему серверу и от них.



Клавиатура

Когда происходит нажатие клавиши, генерируется `keydown` событие type, а когда происходит отпускание клавиши, `keyup` событие. События содержат локализованный символ ключа, например `a`, `s`, `d`или , `f` и код ключа JavaScript,

например `KeyA`, `KeyS`, `KeyD` или `KeyF`. Символы ключей X11 представляют собой шестнадцатеричные сопоставления, определенные в библиотеке Xlib . Некоторые библиотеки JavaScript с открытым исходным кодом, такие как библиотека Keyboard из проекта Guacamole, могут быть использованы для преобразования в коды символов клавиш, необходимые для X11. Затем принимающее приложение может взять эти символы ключей и передать их непосредственно в библиотеку Xlib.

Мышь

События мыши отправляются на экран X11 в виде абсолютного положения или относительного движения. Рассчитав область просмотра клиента браузера, коэффициент масштабирования видео и смещения страницы, вы можете отправить преобразованное событие мыши на экран X11. API PointerLock полезен при попытке захватить указатель мыши и передать относительное (а не абсолютное) движение мыши, что требуется при потоковой передаче 3D нагрузки.

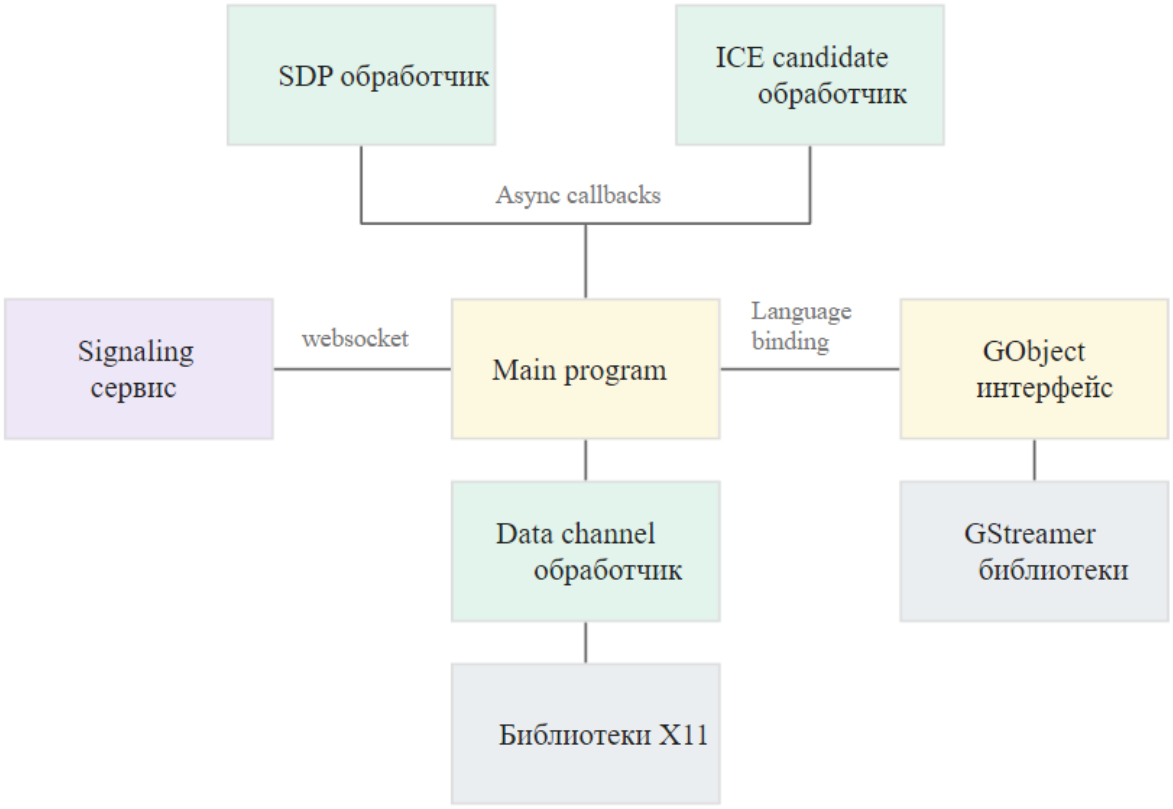
Данные

В дополнение к входным событиям канал данных RTC также может использоваться для отправки командных и управляющих сообщений в конвейер в качестве альтернативы использованию сервера сигнализации. Например, может динамически изменяться битрейт или переключаться видимость указателя мыши. В более сложных случаях использования WebRTC канал данных SCTP заменяется такими протоколами, как Quick UDP Internet Connection (QUIC) .

Приложение GStreamer

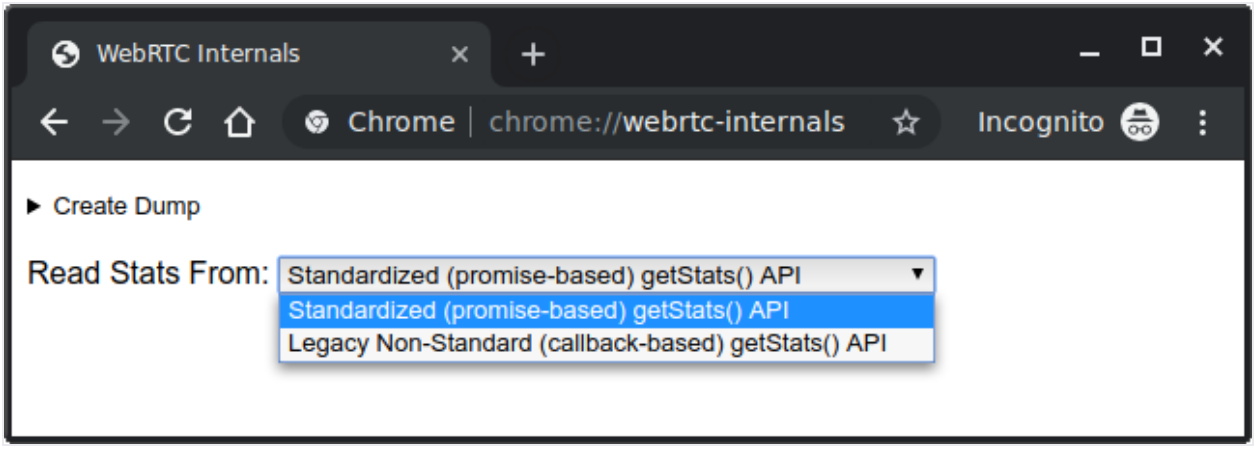
Чтобы использовать `webrtcbin` элемент GStreamer, создается приложение GStreamer, используя одну из поддерживаемых привязок. GStreamer основан на объектной системе GLib (GObject), которая предназначена для языковой совместимости и объектно-ориентированного программирования на языке C. Привязка для языка Python для GStreamer основано на библиотеке PyGObject и является подходящим языком высокого уровня для разработки с использованием современных протоколов и интерфейсов, необходимых приложению WebRTC.

На следующей диаграмме показана архитектура приложения GStreamer WebRTC.



Характеристики и показатели качества соединения WebRTC

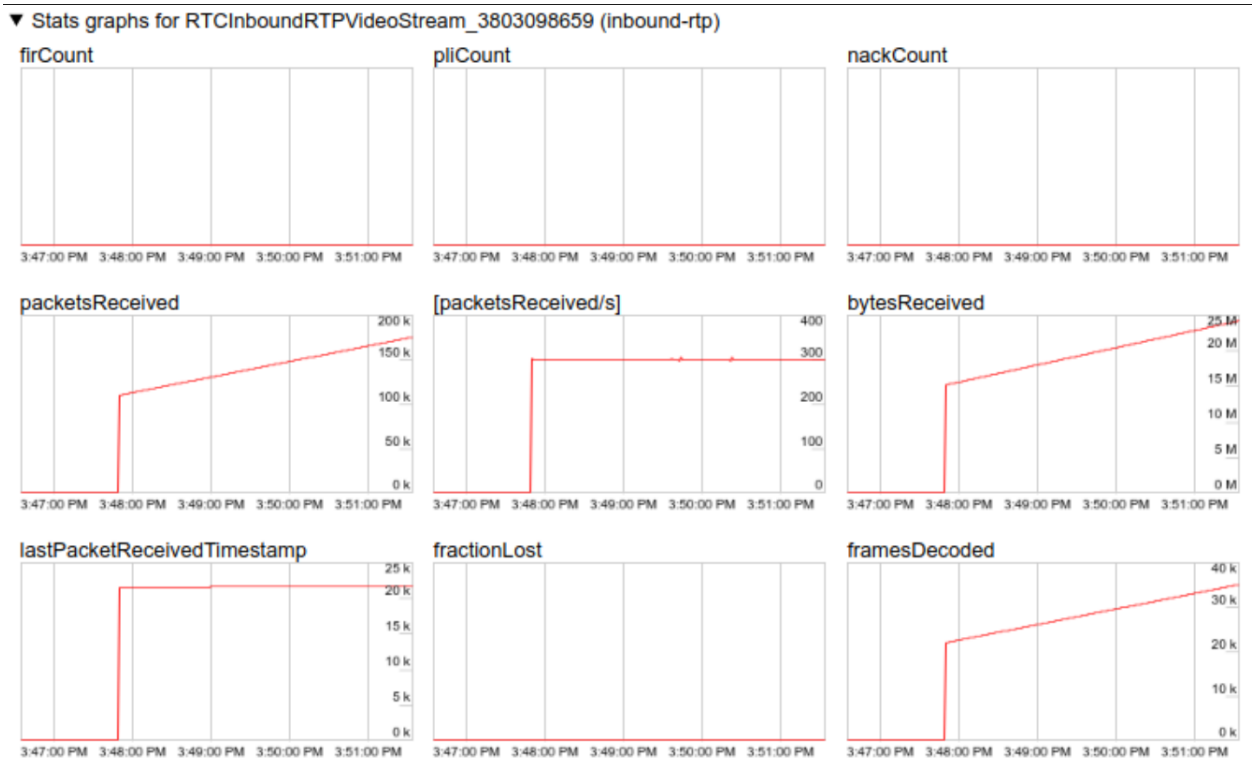
Просмотр `RTCPeerConnection` статистики, возможен по адресу `chrome://webrtc-internals` URI. В настоящее время существует два API статистики: стандартизированный API на основе promises и устаревший API на основе callback для конкретного браузера . В настоящее время в стандартизированном API доступно только набор устаревшей статистики, поэтому в этом примере используется устаревший API.



Эти API предоставляют важную информацию при создании решения на основе WebRTC, например:

- Тип одноранговогo соединения определяется ICE
- Количество принятых и потерянных пакетов
- Задержка соединения
- Производительность буфера джиттера
- Битрейт видео
- Частота кадров видео

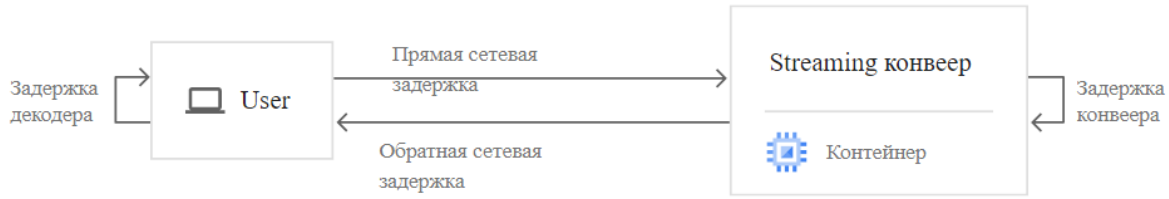
На следующем снимке экрана показаны графики статистики из стандартизированного API.



На следующем снимке экрана показаны графики статистики из устаревшего API.

Задержка

Задержка — важный показатель потоковой передачи, который может значительно повлиять на работу пользователей. Как показано на следующей диаграмме, необходимо учитывать несколько источников задержки, таких как задержка декодирования, задержка конвейера и сетевые задержки в регионах Google Cloud и из них.



Измерение задержки в сети

Сетевую задержку между вашим локальным компьютером источником можно измерить с помощью таких инструментов, как `ping` для измерение задержки туда и обратно между машинами. Использование этой информации позволяет, оценить общую задержку реализации для пользователей. Выбор региона для размещения потоковой рабочей нагрузки также зависит от доступности оборудования в конкретной географической локации

Измерение задержки конвейера

Задержку конвейера можно измерить с помощью трассировщика InterLatency из подключаемого модуля GstShark. Эти метрики позволяют отлаживать и оптимизировать конвейер, определяя медленные элементы. Трассировщик включается путем установки плагина, а затем запуска конвейера с необходимыми переменными среды:

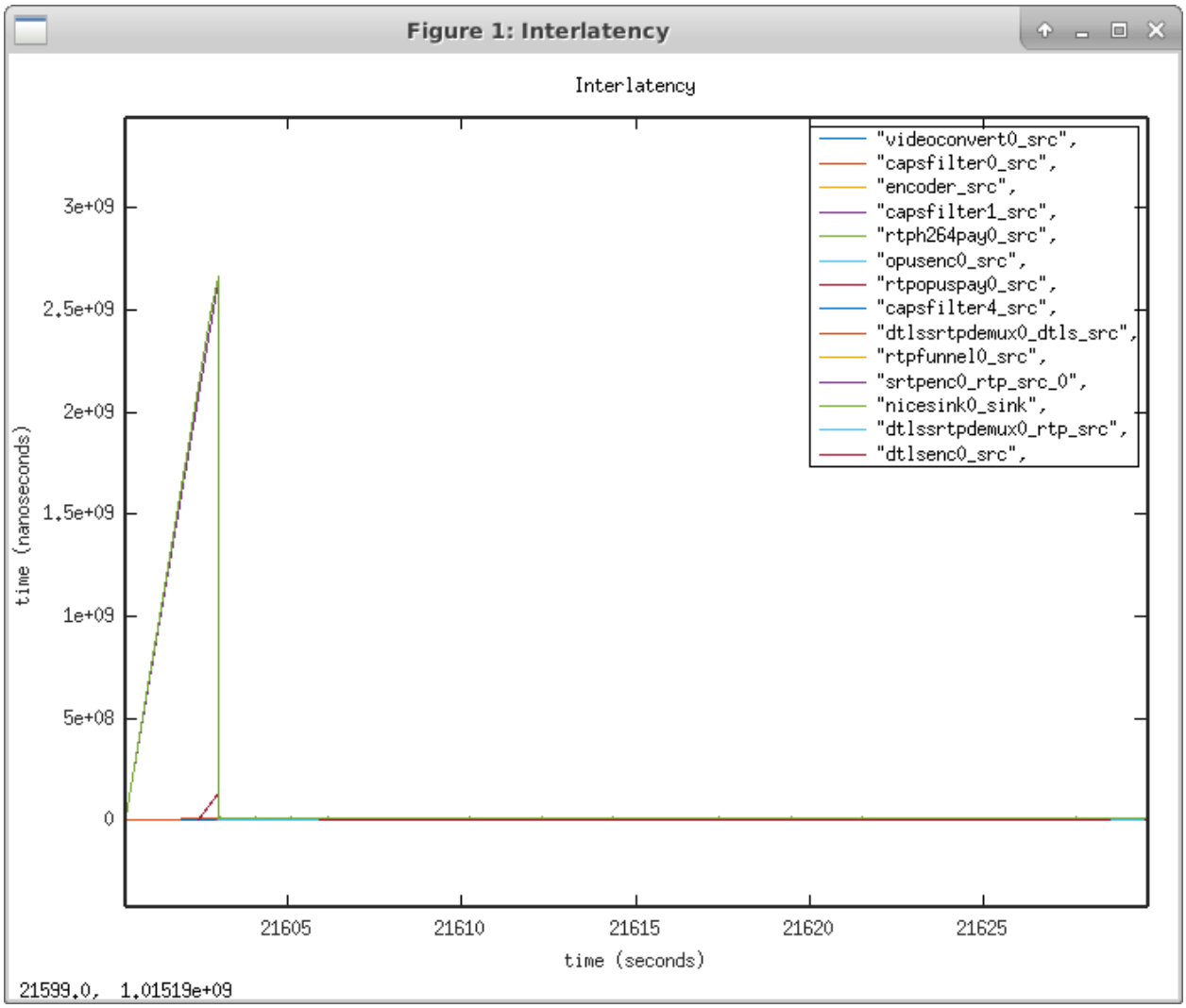
```
GST_DEBUG="GST_TRACER:7"
GST_TRACERS="interlatency"
```

Когда конвейер корректно останавливается `SIGINT` сигналом, создаются каталог и файлы данных:

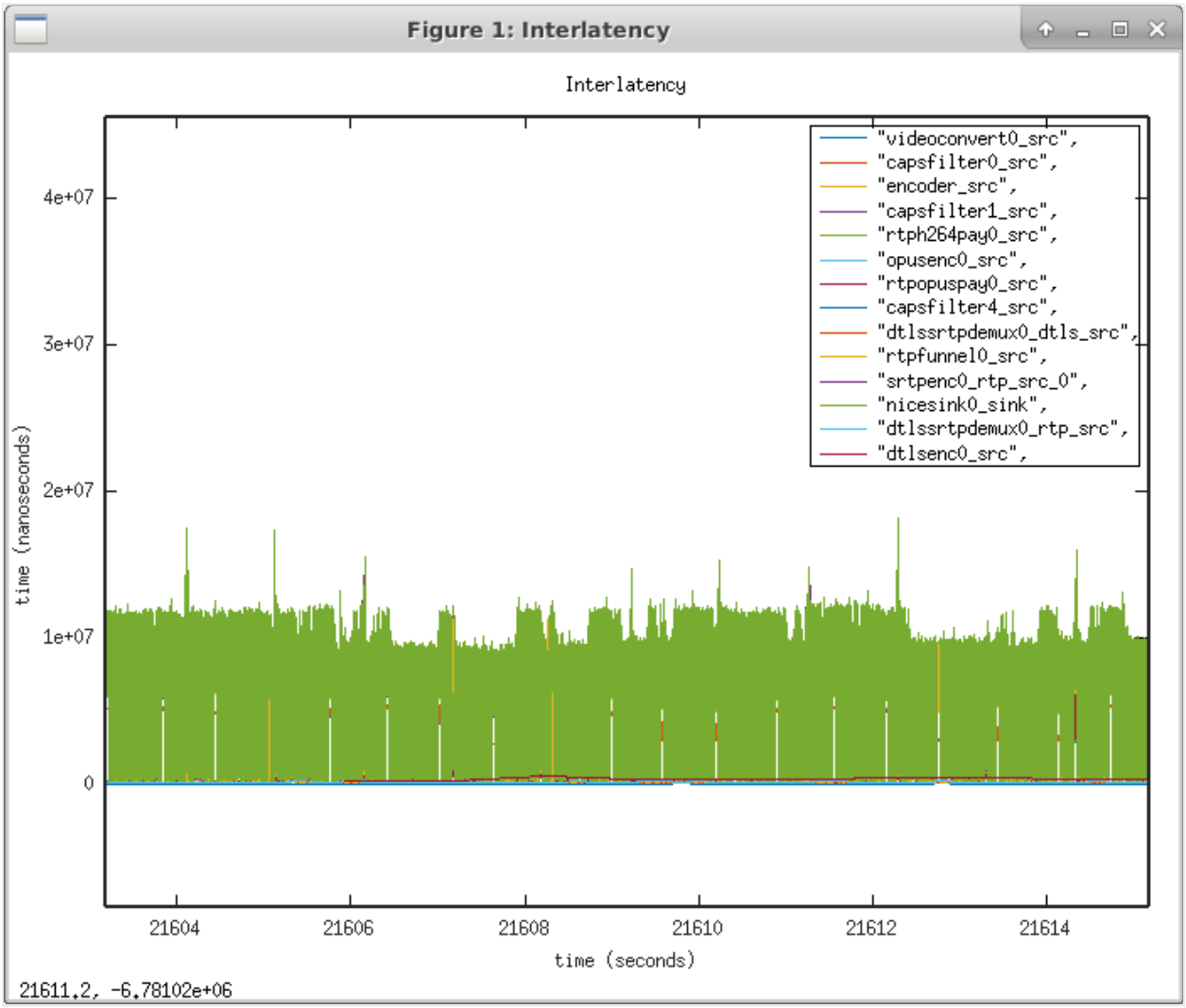
```
gstshark_DATE_TIME/
├── datastream
└── metadata
```

Этот каталог передается в качестве аргумента скрипту `gstshark-plot` , который генерирует интерактивный график с помощью Octave . График позволяет исследовать задержку, измеренную во время работы конвейера. Первоначальный запуск имеет большую длительность, поскольку конвейер инициализируется, но в конечном итоге достигает устойчивого состояния. Задержка кодирования также является важным составляющей суммарной задержки. В этом примере он добавляет от 5 до 12 мс, как показано на следующих снимках экрана.

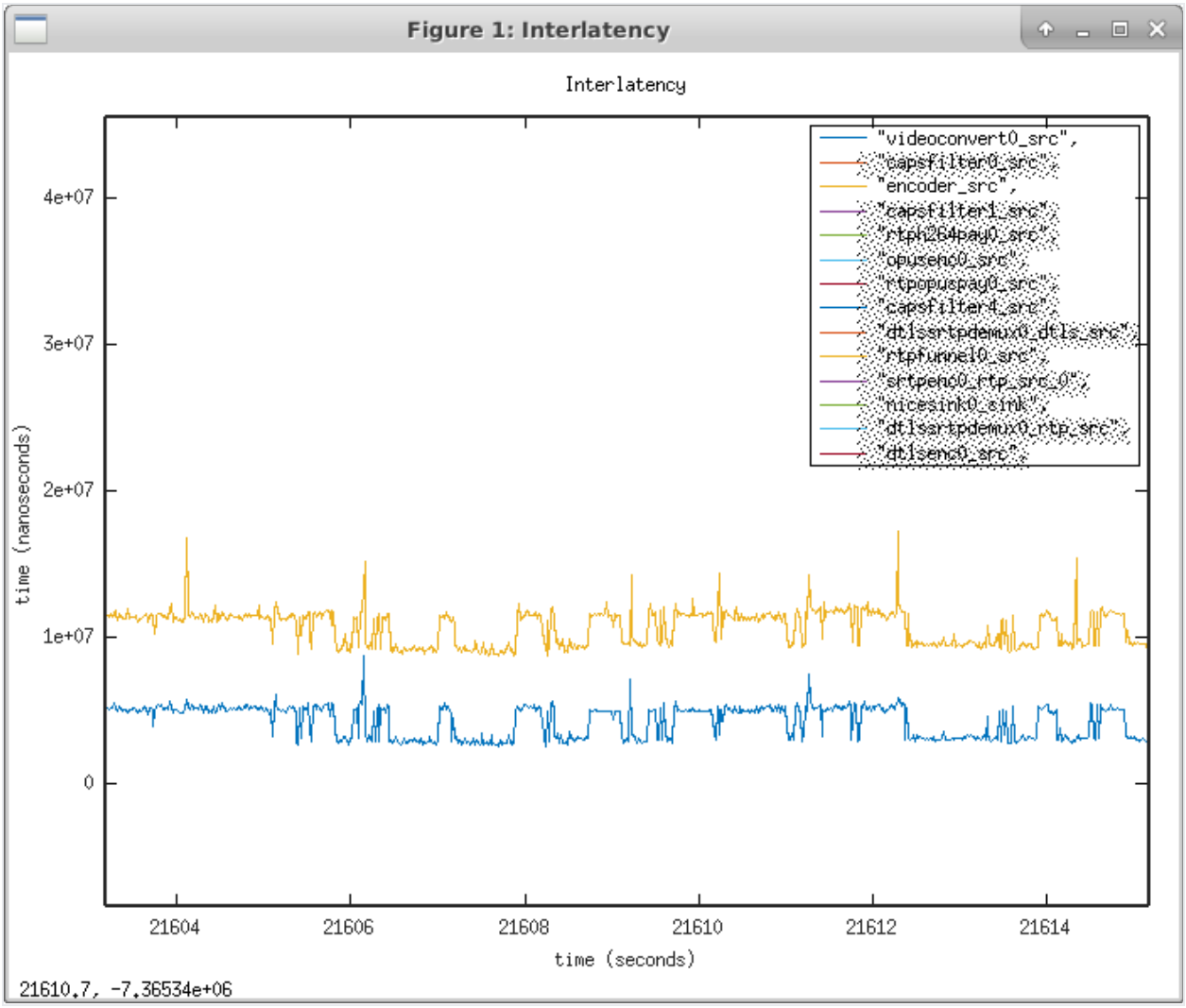
На рисунке ниже показан график внутренней задержки конвейера, показывающий начальный запуск.



На следующем рисунке показана внутренняя задержка конвейера в устойчивом состоянии для всех элементов конвейера.



И, наконец, на следующем рисунке показана взаимосвязь кодирования видео и преобразования формата.

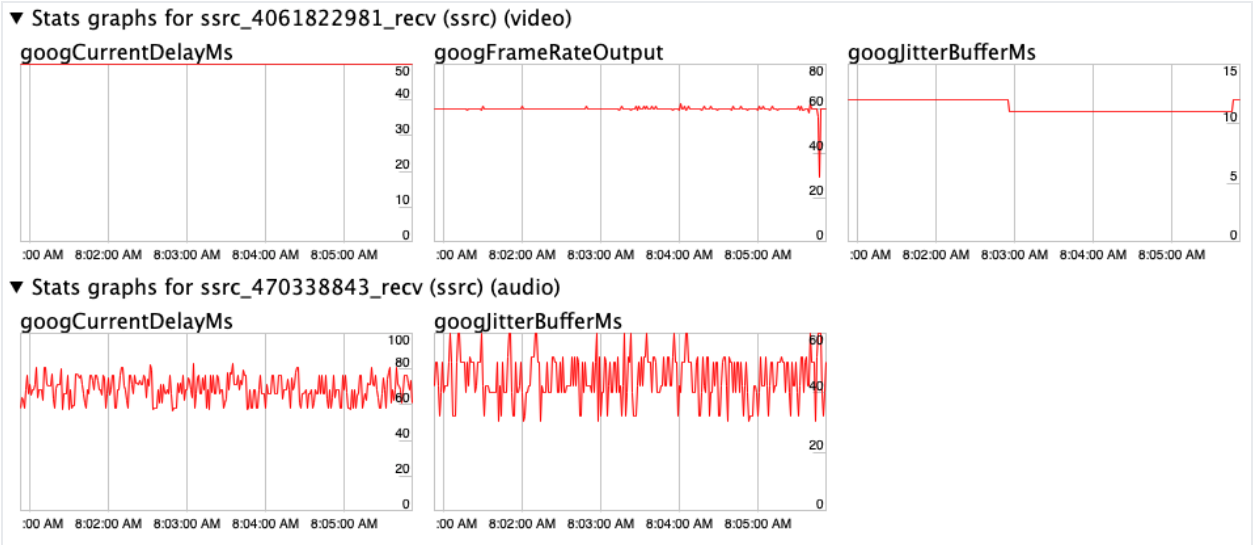


Измерение задержки клиента в Chrome

Производительность подключения наиболее удобно контролировать с помощью `chrome://webrtc-internals` страницы. На задержку влияют следующие ключевые показатели:

- Видео `googCurrentDelayMs`
- Видео `googJitterBufferMs`
- Видео `googFrameRateOutput`
- Аудио `googCurrentDelayMs`
- Аудио `googJitterBufferMs`

На следующем рисунке показана общая воспринимаемая пользователем задержка.



Задержка представляет собой комбинацию задержки видео и аудио. Уменьшение обоих этих факторов позволяет создать лучший пользовательский опыт.

WebRTC использует буферы Jitter для хранения пакетов между поступлением и воспроизведением. Эти буферы используются для повторной сборки неупорядоченных и задержанных пакетов. Чем меньше буфер, тем меньше задержка. Клиентское ПО динамически изменяет размер буферов Jitter с помощью NetEQ (для звука), чтобы адаптироваться к условиям сети. Если сервер, отправляющий поток, работает медленно или имеет плохое сетевое соединение, буфер Jitter увеличивается до тех пор, пока не достигнет максимального размера около 1000 мс. Чтобы уменьшить этот эффект, отправляющий поток может отбрасывать аудио-фрагменты или видеокадры, чтобы задержанные пакеты никогда не отправлялись.

Если сервер-отправитель испытывает высокую нагрузку на CPU, это также может замедлить конвейер и вызвать задержку.

Исправление проблем

WebRTC — это комплексный протокол, включающий множество технологий, которые могут быть сложными для отладки и отслеживания проблем. Самая распространенная проблема — подключение. Обход брандмауэров и пользовательских сетей усложняется и усложняет отладку других проблем с конвейером GStreamer. В этом разделе перечислены основные проблемы,

которые возникают при создании потокового приложения на основе WebRTC, и способы их отладки.

Трассировка подключения

Пример Trickle ICE , созданный командой WebRTC, является полезным инструментом для проверки подключения из вашего локального браузера к серверу STUN или TURN. Требуется ввести адрес и учетные данные сервера TURN и нажать «Gather candidates» , чтобы проверить подключение. Адрес находится в форме URI, переданного в вашу `RTCPeerConnection` конфигурацию, например:

```
stun:5.4.3.2:3478
turn:5.4.3.2:3478?transport=udp
```

Если проверка прошла успешно, появляется запись в журнале ICE, содержащая IP-адрес сервера TURN. Если IP-адрес отсутствует, вероятно, проблема связана с брандмауэром или сервером TURN.

Time	Component	Type	Foundation	Protocol	Address	Port	Priority
0,002	1	host	619629460	UDP	192.168.1.2	50589	126 | 30 | 255
0,047	1	srflx	842163049	UDP	1.2.3.4	40326	100 | 30 | 255
0,104	Done						
0,106							

Отслеживание установления соединения

Описанная ранее процедура установления соединения WebRTC имеет несколько состояний, через которые соединение должно пройти, прежде чем оно будет

установлено. Страница chrome://webrtc-internals является одним из ресурсов для отслеживания этого процесса.

На следующем снимке экрана показан пример вызова трассировки WebRTC API.

Time	Event
7/7/2019, 4:41:57 PM	▶ setRemoteDescription
7/7/2019, 4:41:57 PM	▶ addIceCandidate (host)
7/7/2019, 4:41:57 PM	▶ addIceCandidate (host)
7/7/2019, 4:41:57 PM	▶ addIceCandidate (host)
7/7/2019, 4:41:57 PM	▶ addIceCandidate (host)
7/7/2019, 4:41:57 PM	▶ addIceCandidate (host)
7/7/2019, 4:41:57 PM	▶ addIceCandidate (host)
7/7/2019, 4:41:57 PM	▶ addIceCandidate (srflx)
7/7/2019, 4:41:57 PM	▶ addIceCandidate (srflx)
7/7/2019, 4:41:57 PM	▶ transceiverAdded
7/7/2019, 4:41:57 PM	▶ transceiverAdded
7/7/2019, 4:41:57 PM	▶ signalingstatechange
7/7/2019, 4:41:57 PM	setRemoteDescriptionOnSuccess
7/7/2019, 4:41:57 PM	▶ addIceCandidate (srflx)
7/7/2019, 4:41:57 PM	▶ createAnswer
7/7/2019, 4:41:57 PM	▶ addIceCandidate (srflx)
7/7/2019, 4:41:57 PM	▶ addIceCandidate (srflx)
7/7/2019, 4:41:57 PM	▶ addIceCandidate (srflx)
7/7/2019, 4:41:57 PM	▶ addIceCandidate (relay)
7/7/2019, 4:41:57 PM	▶ addIceCandidate (relay)
7/7/2019, 4:41:57 PM	▶ addIceCandidate (relay)
7/7/2019, 4:41:57 PM	▶ addIceCandidate (relay)
7/7/2019, 4:41:57 PM	▶ createAnswerOnSuccess
7/7/2019, 4:41:57 PM	▶ setLocalDescription
7/7/2019, 4:41:57 PM	▶ transceiverModified
7/7/2019, 4:41:57 PM	▶ transceiverModified
7/7/2019, 4:41:57 PM	▶ signalingstatechange
7/7/2019, 4:41:57 PM	setLocalDescriptionOnSuccess
7/7/2019, 4:41:57 PM	▶ icegatheringstatechange
7/7/2019, 4:41:57 PM	▶ iceconnectionstatechange
7/7/2019, 4:41:57 PM	▶ icecandidate (host)
7/7/2019, 4:41:57 PM	▶ icecandidate (srflx)
7/7/2019, 4:41:57 PM	▶ icecandidate (relay)
7/7/2019, 4:41:57 PM	▶ icecandidate (relay)
7/7/2019, 4:41:57 PM	▶ icegatheringstatechange
7/7/2019, 4:41:57 PM	▶ iceconnectionstatechange
7/7/2019, 4:41:59 PM	▶ onRemoteDataChannel

Пример трассировки WebRTC API

Отладка конвейера GStreamer

Конвейер GStreamer состоит из нескольких отдельных элементов, каждый из которых имеет собственные журналы отладки и уровни журналов.

Установка уровня журнала для всех элементов, задается переменной оболочки `GST_DEBUG` перед запуском конвейера. Значение этой переменной имеет следующий формат:

```
GST_DEBUG=element:level
```

В следующем примере кода уровень журнала устанавливается равным 3 для всех элементов:

```
GST_DEBUG=*:3
```

Уровень 3 предназначен для обнаружения высокоуровневой информации и фатальных ошибок.

Чтобы увидеть более подробные журналы для `nvenc` элемента используется команда:

```
GST_DEBUG=nvenc:4
```

Может быть запущен любой раздел конвейера в командной строке вне приложения GStreamer Python с помощью команды `gst-launch-1.0`. Этот `fakesink` элемент полезен для отладки, так как он выводит в пустой приемник. Этот инструмент полезен для быстрой проверки работоспособности и проверки правильности установки библиотек GStreamer и драйверов NVENC.

```
GST_DEBUG=ximagesrc,nvenc:4 \
  gst-launch-1.0 \
  ximagesrc use-damage=0 remote=1 ! \
  videoconvert ! \
  nvh264enc ! \
  fakesink num-buffers=1
```

Ссылки:

1. GStreamer - library for constructing graphs of media-handling components. (https://gstreamer.freedesktop.org/)

2. WebRTC - Real-time communication for the web (https://webrtc.org/)
3. Введение в протоколы WebRTC (https://developer.mozilla.org/ru/docs/Web/API/WebRTC_API/Protocols)