Connection patterns

- Before you start
  - Network connectivity
- Publish audio or video
- Receive a published track
- Send messages between two endpoints
- Receive media and send controls
- Add media to an existing connection
- Add DataChannels to a media connection
- Other initial negotiation sequences
- Complete an SFU offer
- Close media tracks
  - Close with negotiation
  - Stop forwarding without negotiation
- Clean up

import { Steps } from "~/components";

Choose the task for your endpoint. Each initial setup recipe starts with a new browser connection and matching SFU session. Your application can combine these endpoints into rooms, broadcasts, and device interfaces.

Create the SFU session when ready to begin that connection's SDP exchange. Unconnected sessions can expire before their first operation.

Task	Result	Runnable example
Publish audio or video	Make local media available to subscribers	Video room
Receive a published track	Play another endpoint's media	Video room
Send messages between two endpoints	Exchange application data without media tracks	DataChannel example

Task	Result	Runnable example
Receive media and send controls	Watch a publisher and return authorized input	Cloud gaming

For an existing connection, refer to Add media, Add DataChannels, or Close media tracks. To run an application before implementing these exchanges, follow the quickstart.

Before you start

Create an SFU app. Your backend holds its App Secret and makes every SFU API request shown here. Use Connection API authentication. Paths are relative to `https://rtc.live.cloudflare.com/v1/apps/{appId}`.

In a browser, an `RTCPeerConnection` manages one connection to the SFU. The matching `sessionId` identifies that connection in API calls. Session Description Protocol (SDP) describes the connection's media and transport settings. The browser and SFU exchange an offer and answer to agree on those settings.

Check the HTTP status, public errors, and individual resource results on every API call. Retain successful allocations for cleanup. Complete each request and its required SDP exchange before starting the next mutation on the same session. Different sessions can proceed independently. Negotiation and session lifecycle explains how to coordinate overlapping handlers.

Network connectivity

WebRTC uses Interactive Connectivity Establishment (ICE) to find a working network path. Configure it for the networks your endpoints support. Cloudflare's public STUN server is `stun.cloudflare.com:3478`. STUN helps discover network addresses. TURN can relay traffic when an endpoint's network requires it.

Install connection-state handlers and give connection waits a timeout that reports failure. After setting a local description, gather network candidates before sending `pc.localDescription` to your backend. The cloud-gaming WebRTC helpers demonstrate bounded candidate gathering.

Publish audio or video

This recipe publishes a browser camera or microphone. It creates a media connection without DataChannel transport.

1. **Prepare the media offer.** In the browser, capture tracks with `navigator.mediaDevices.getUserMedia()`. Create an `RTCPeerConnection`, called `pc` here, and add each track with `pc.addTransceiver(track, { direction: "sendonly" })`. Create an offer with `await pc.createOffer()`, then apply it with `await pc.setLocalDescription(offer)`.

   Wait for candidate gathering, then read each transceiver's `mid`. This identifier associates a media section in this connection's SDP with the track you publish.
2. **Create the SFU session.** On your backend, call `POST /sessions/new` with no request body. Store the returned `sessionId` with this connection.
3. **Publish the tracks.** On the backend, call `POST /sessions/{sessionId}/tracks/new` with the browser's `pc.localDescription` as `sessionDescription`:

   ```
   {
     "sessionDescription": { "type": "offer", "sdp": "<LOCAL_SDP>" },
     "tracks": [
       { "location": "local", "mid": "<TRANSCEIVER_MID>", "trackName": "camera" }
     ]
   }
   ```

   Replace the placeholders with the generated SDP and assigned `mid`. To publish camera and microphone together, include both tracks in this request. Each entry uses its own `mid` and `trackName`. Refer to batch resource operations.
4. **Apply the answer.** In the browser, await `pc.setRemoteDescription(response.sessionDescription)` with the returned SFU answer. This completes the offer/answer exchange. No `/renegotiate` call is needed for this answer.
5. **Make the publication discoverable.** Wait for the connection to become `connected`. Share successful publications' session ID and track names with authorized subscribers through your application.

The publication is ready for another endpoint to request. Use Receive a published track to verify playback.

Receive a published track

Start with the publisher's session ID and track name, obtained through your application. This receiving-only browser does not need camera or microphone access.

1. **Create the receiving connection.** Create a new `RTCPeerConnection`, called `pc`, and install a `track` event handler. On the backend, call `POST /sessions/new` with no request body. Save its ID as `subscriberSessionId`.

2. **Request the publication.** On the backend, call `POST /sessions/{subscriberSessionId}/tracks/new` without a local offer:

```
{
  "tracks": [
    { "location": "remote", "sessionId": "<PUBLISHER_SESSION_ID>", "trackName":
"camera" }
  ]
}
```

   To receive several publications together, include one entry for each in `tracks`. They can come from different publishers. This request adds them all to the same receiving session.
3. **Answer the SFU offer.** Map each successful result's receiving `mid` to its publication before applying the returned offer. In the browser, await `pc.setRemoteDescription(response.sessionDescription)`, then `pc.createAnswer()` and `pc.setLocalDescription(answer)`.
   In the `track` handler, use `event.transceiver.mid` to select the publication's playback element. For one received track, set `element.srcObject = new MediaStream([event.track])`.
4. **Finish the exchange.** After candidate gathering, return `pc.localDescription` to your backend. Send it as `sessionDescription` to `PUT /sessions/{subscriberSessionId}/renegotiate`:

```
{
  "sessionDescription": { "type": "answer", "sdp": "<LOCAL_ANSWER_SDP>" }
}
```

   Continue only after the API accepts the answer.
5. **Verify reception.** Wait for the connection to become `connected` and confirm that the requested media plays. Use media troubleshooting when the connection succeeds but playback does not.

Send messages between two endpoints

This recipe connects a publisher and a subscriber without audio or video. It uses reliable, ordered delivery. First connect each endpoint's DataChannel transport, then allocate the application channel carried over it.

1. **Create both connections.** In each browser, create a new `RTCPeerConnection`. Install a `datachannel` event listener to retain channels opened by the SFU and close them during

teardown. On the backend, call `POST /sessions/new` with no body for each connection. Save `publisherSessionId` and `subscriberSessionId`.

2. **Connect DataChannel transport.** For each session, call `POST /sessions/{sessionId}/datachannels/establish` once:

```
{
  "dataChannel": {
    "location": "remote",
    "dataChannelName": "server-events"
  }
}
```

   Check the returned `dataChannel` allocation. The `server-events` channel is reserved for transport setup. The application channel is allocated separately.

3. **Answer each SFU offer.** In the corresponding browser, await `pc.setRemoteDescription(response.sessionDescription)`, create an answer, and await `pc.setLocalDescription(answer)`. After candidate gathering, return `pc.localDescription` to your backend and send `PUT /sessions/{sessionId}/renegotiate`:

```
{
  "sessionDescription": { "type": "answer", "sdp": "<THIS_ENDPOINT_ANSWER_SDP>" }
}
```

   Wait for each answer to be accepted and both PeerConnections to become `connected` before allocating application channels.

4. **Publish the application channel.** On the backend, call `POST /sessions/{publisherSessionId}/datachannels/new`:

```
{
  "dataChannels": [{ "location": "local", "dataChannelName": "messages" }]
}
```

   Retain the successful publisher allocation and its `id`.

5. **Subscribe to the channel.** Call `POST /sessions/{subscriberSessionId}/datachannels/new`:

```
{
  "dataChannels": [
    {
      "location": "remote",
      "sessionId": "<PUBLISHER_SESSION_ID>",
      "dataChannelName": "messages"
    }
  ]
}
```

Retain the successful subscriber allocation and its `id`.

6. **Create the browser channels.** Return each allocation's `id` to its own endpoint as `channelId`. On each PeerConnection, create the channel with that ID:

   ```
   const channel = pc.createDataChannel("messages", {
     negotiated: true,
     id: channelId,
   });
   ```

   Publisher and subscriber IDs can differ. This browser configuration matches the reliable, ordered API defaults. Install a `message` handler on the subscriber and wait for both channels to open.
7. **Send and verify.** After the subscriber confirms it is ready through your application, call `channel.send("hello")` on the publisher. Confirm the subscriber's `message` event contains `hello`.

You now have publisher-to-subscriber messages. To add a return path, use DataChannel replies. For another delivery policy, publish a separate named channel and have its subscribers mirror that policy. Refer to delivery settings.

Receive media and send controls

This recipe uses one browser connection for incoming media and outgoing controls. The publisher already supplies a media track and a reliable, ordered `controls` DataChannel. Your backend has both publication locators and has authorized this browser as the controller.

1. **Create the receiving connection.** Create a new `RTCPeerConnection`, called `pc`. Install `track` and `datachannel` listeners. On the backend, call `POST /sessions/new` with no body and save `subscriberSessionId`. Do not create browser DataChannels during this first media exchange.
2. **Request media.** Call `POST /sessions/{subscriberSessionId}/tracks/new` without a local offer:

   ```
   {
     "tracks": [
       { "location": "remote", "sessionId": "<MEDIA_PUBLISHER_SESSION_ID>", "trackName":
   "camera" }
     ]
   }
   ```

   Map successful receiving `mid` values to their publications. Use `event.transceiver.mid` in the `track` handler to attach each received track to its playback element.

3. **Finish media negotiation.** Await `pc.setRemoteDescription(response.sessionDescription)`, `pc.createAnswer()`, and `pc.setLocalDescription(answer)`. After candidate gathering, send `pc.localDescription` through your backend to `PUT /sessions/{subscriberSessionId}/renegotiate`:

```
{
  "sessionDescription": { "type": "answer", "sdp": "<MEDIA_ANSWER_SDP>" }
}
```

   Wait for the API to accept it and confirm media playback before continuing.
4. **Add DataChannel transport.** Keep this same PeerConnection and session. Call `POST /sessions/{subscriberSessionId}/datachannels/establish` once:

```
{
  "dataChannel": {
    "location": "remote",
    "dataChannelName": "server-events"
  }
}
```

   Retain the successful allocation and reserved `server-events` channel for cleanup.
5. **Finish the new negotiation.** Apply this new SFU offer with `pc.setRemoteDescription()`. Create and set a new local answer. After candidate gathering, submit `pc.localDescription` through your backend to `PUT /sessions/{subscriberSessionId}/renegotiate`:

```
{
  "sessionDescription": { "type": "answer", "sdp": "<DATACHANNEL_ANSWER_SDP>" }
}
```

   Await each browser operation and the API's acceptance of the answer. Wait for the PeerConnection to be `connected` before allocating the control subscription.
6. **Subscribe with reply permission.** Call `POST /sessions/{subscriberSessionId}/datachannels/new`:

```
{
  "dataChannels": [
    {
      "location": "remote",
      "sessionId": "<CONTROL_PUBLISHER_SESSION_ID>",
      "dataChannelName": "controls",
      "canReply": true
    }
  ]
}
```

Check the result. At most one subscriber can hold reply access for this publisher channel. Granting access to this subscriber replaces any previous holder. Your backend controls who receives that permission.

7. **Open and verify controls.** Return the subscription's `id` to the browser as `channelId`:

   ```
   const controls = pc.createDataChannel("controls", {
     negotiated: true,
     id: channelId,
   });
   ```

   Wait for `open`. Send a command defined by your application and confirm that the publisher receives it while media continues playing.

The cloud-gaming guide shows media and input on one browser connection, including controller ownership and separate delivery settings for discrete and replaceable input.

Add media to an existing connection

Start with a connected PeerConnection and keep its SFU session. Serialize additions with other operations on that session.

- **Publish more media:** Add transceivers for the new tracks, then create and set a fresh local offer. Follow the publication recipe from `tracks/new`, sending the new local entries with the gathered `pc.localDescription`.
- **Receive more media:** Call `tracks/new` on the same session with additional remote entries in the receive request format. Complete any returned SFU offer.

Finish the required SDP exchange before the next mutation.

Add DataChannels to a media connection

Start here when an existing PeerConnection has completed media-only negotiation and has no DataChannel transport. Keep its SFU session and finish any outstanding operation first.

1. Install a `datachannel` listener to retain the reserved channel for teardown. Call `POST /sessions/{sessionId}/datachannels/establish` with `dataChannel: { location: "remote", dataChannelName: "server-events" }`.
2. Check the allocation and apply the returned offer. Await creation and local application of a new answer. After candidate gathering, submit `pc.localDescription` through your backend to `PUT /sessions/{sessionId}/renegotiate` as `sessionDescription`.
3. Wait for API acceptance and a `connected` PeerConnection. You can now add application channels using `datachannels/new`.

Other initial negotiation sequences

The recipes use a known starting state. Existing integrations may negotiate transport in a different order:

- **DataChannels before media:** complete the message recipe's transport exchange, then add media on the same session. Finish each exchange before the next operation.
- **Media and DataChannels in the first offer:** an endpoint can negotiate both in its initial `tracks/new` offer. After that exchange, skip `datachannels/establish` and allocate application channels through `datachannels/new`. The Pocket Radio firmware walkthrough demonstrates this setup.
- **DataChannel setup already started:** finish its outstanding offer/answer exchange and connection wait. An already-negotiated transport can carry additional application channels. Do not use `datachannels/establish` as a repeated initialization check.

Complete an SFU offer

An operation returning `sessionDescription.type: "offer"` with `requiresImmediateRenegotiation: true` needs an answer before the next mutation on that session. The recipes include this exchange. Use this reference when implementing another operation.

1. In the browser, await `pc.setRemoteDescription(response.sessionDescription)`.
2. Await `pc.createAnswer()` and `pc.setLocalDescription(answer)`. After candidate gathering, return `pc.localDescription` to your backend.
3. Send `PUT /sessions/{sessionId}/renegotiate` with `sessionDescription: { type: "answer", sdp: "<LOCAL_ANSWER_SDP>" }`.
4. Check that the API accepted the answer before starting another mutation. A browser's `signalingState` becoming `stable` does not confirm that the backend has submitted the answer.

Close media tracks

Use the `mid` from the connection you are modifying. Closing a receiving track stops that subscription. Closing a publishing track stops the source. A successful close of a bidirectional transceiver stops both directions. Subscribers' PeerConnections can remain connected, and media already buffered can still play.

Close with negotiation

Start with an established connection and no unfinished offer/answer exchange.

1. **Prepare the endpoint's offer.** In the browser, save the `mid` of each transceiver to close, then call its `stop()` method. Create an offer with `await pc.createOffer()` and apply it with `await pc.setLocalDescription(offer)`.
2. **Close the tracks.** After candidate gathering, send `PUT /sessions/{sessionId}/tracks/close` from your backend with those mids and the browser's `pc.localDescription` as `sessionDescription`:

   ```
   {
     "tracks": [{ "mid": "<TRANSCEIVER_MID>" }],
     "sessionDescription": { "type": "offer", "sdp": "<LOCAL_SDP>" },
     "force": false
   }
   ```

3. **Apply the answer.** Check request-level and per-track errors. In the browser, await `pc.setRemoteDescription(response.sessionDescription)` with the returned answer, even if individual tracks failed. Other tracks may have closed.
4. **Handle unfinished work.** Complete this exchange before another mutation. Retain failed closures for retry. If no usable answer arrives, follow recovery guidance.

Use close results to distinguish successful closure, an already-closed track, and an unresolved failure.

Stop forwarding without negotiation

Set `force: true` to close selected media flows without exchanging SDP. This is useful when the endpoint cannot supply a close offer. On the backend, send `PUT /sessions/{sessionId}/tracks/close`:

```
{
  "tracks": [{ "mid": "<TRANSCEIVER_MID>" }],
```

```
  "force": true
}
```

Check each track result. No `/renegotiate` call is needed. Forced closure can leave records in session information. Follow teardown ordering for outstanding negotiation or requests.

Clean up

Stop accepting new operations and withdraw ended publications from discovery. Close allocated media tracks and DataChannels, then close the endpoint's PeerConnection and stop captures when no longer needed. Follow cleanup responsibilities if the endpoint leaves before finishing.

□□□ #2
□ □□□ □□□ 2026-11-01 03:45:50 UTC
□ □□□ □□□ 2026-11-01 03:45:52 UTC