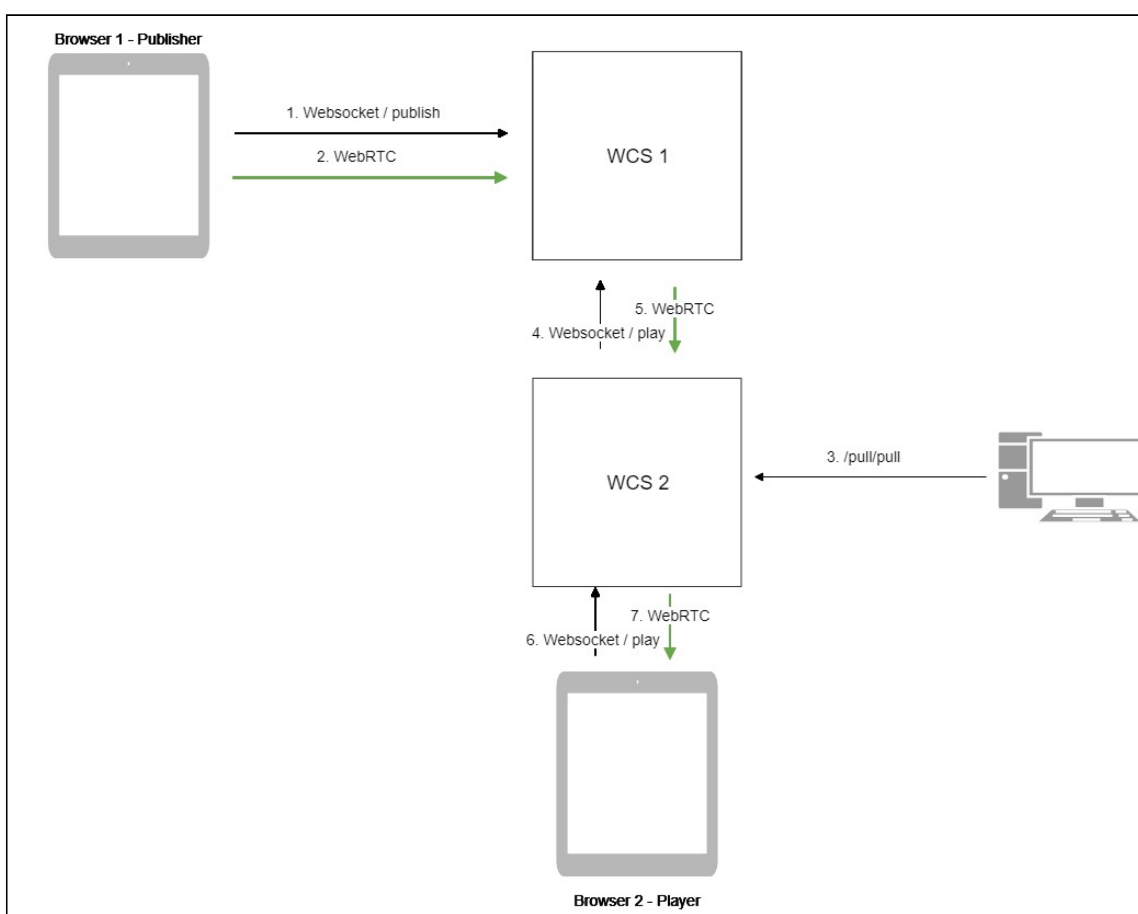
From another WCS server via WebRTC

Overview

WCS can capture on demand a WebRTC video stream published by another WCS server. The captured stream can be than played on any of supported platforms using any of supported technologies. WebRTC stream capturing is managed using REST API.

Operation flowchart



1. The browser connects to the WCS1 server via Websocket and sends the `publishStream` command.
2. The browser captures the microphone and the camera and sends the WebRTC stream to the server.
3. The REST client sends to the WCS2 server the `/pull/pull` query.
4. WCS2 requests the stream from WCS1.

5. WCS2 receives the WebRTC stream from WCS1.
6. The second browser establishes a connection to the WCS2 server via Websocket and sends the `playStream` command.
7. The second browser receives the WebRTC stream and plays this stream on the page.

REST API

REST query must be an HTTP/HTTPS POST request as follows:

- HTTP: `http://test.flashphoner.com:8081/rest-api/pull/rtmp/pull`
- HTTPS: `https://test.flashphoner.com:8444/rest-api/pull/rtmp/pull`

Where:

- `test.flashphoner.com` is the address of the WCS server
- `8081` is the standard REST / HTTP port of the WCS server
- `8444` is the standard HTTPS port
- `rest-api` is the required part of the URL
- `/pull/rtmp/pull` is the REST method used

REST queries and responses

REST method	Request body	Response body	Response status	Description

REST method	Request body	Response body	Response status	Description
\`/pull/pull\`	`{ "uri":"wss://demo.flashphoner.com:8443", "localStreamName": "testStream", "remoteStreamName": "testStream" }`		409 Conflict 500 Internal error	Pull the WebRTC stream at the specified URL

<table>
<tr><th>REST method</th><th>Request body</th><th>Response body</th><th>Response status</th><th>Description</th></tr>
<tr><td>`/pull/find_all`</td><td></td><td>

```
[
    {
        "localMediaSessionId": "5a072377-73c1-4caf-abd3",
        "remoteMediaSessionId": null,
        "localStreamName": "testStream",
        "remoteStreamName": "testStream",
        "uri": "wss://demo.flashphoner.com:8443",
        "status": "NEW"
    }
]
```

</td><td>200 OK 404 Not found 500 Internal error</td><td>Find all pulled WebRTC streams</td></tr>
</table>

REST method	Request body	Response body	Response status	Description
`/pull/terminate`	`{ "uri":"wss://demo.flashphoner.com:8443", "localStreamName": "testStream", "remoteStreamName": "testStream" }`		200 OK 404 Not found 500 Internal error	Terminate the pulled WebRTC stream

Parameters

Parameter	Description	Example
uri	Websocket URL of WCS server	`wss://demo.flashphoner.com:8443`
localMediaSessionId	Session identifier	`5a072377-73c1-4caf-abd3`
remoteMediaSessionId	Session identifier on the remote server	`12345678-abcd-dead-beaf`
localStreamName	Local name assigned to the captured stream. By this name the stream can be requested from the WCS server	`testStream`
remoteStreamName	Captured stream name on the remote server	`testStream`
status	Current stream status	`NEW`

Configuration

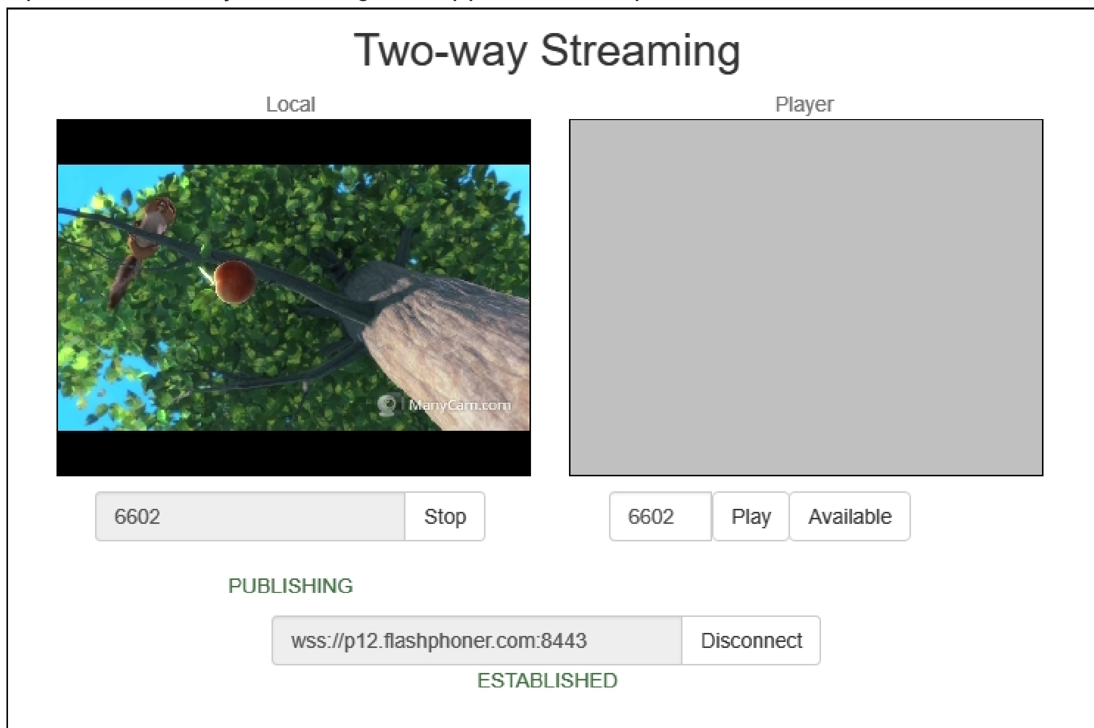
By default, WebRTC stream is pulled over unsecure Websocket connection, i.e. WCS server URL has to be set as `ws://demo.flashphoner.com:8080`. To use Secure Websocket, the parameter must be set in flashphoner.properties file

```
wcs_agent_ssl=true
```

This change has to be made on both WCS servers: the server that publishes the stream and the server the stream is pulled to.

Quick manual on testing

1. For this test we use:
2. two WCS servers;
3. the Chrome browser and a REST-client to send queries to the server;
4. the Two Way Streaming web application to publish the stream;
5. the Player web application to play the captured stream in the browser.
6. Open the Two Way Streaming web application and publish the stream on the first server



7. Open the REST client. Send the `/pull/pull` query to the second WCS server and specify the following parameters:
8. URL of the WCS server the stream is captured from;
9. stream name published on the server;

10. local stream name

Method
POST

Request URL
http://p11.flashphoner.com:9091/rest-api/pull/pull

SEND

Parameters

Headers | Body | Variables

Body content type
application/json

Editor view
Raw input

FORMAT JSON MINIFY JSON

```
{
  "uri": "wss://p12.flashphoner.com:8443",
  "remoteStreamName": "6602",
  "localStreamName": "6602"
}
```

200 OK 93.10 ms DETAILS

11. Make sure the second server captured the stream. To do this, send the `/pull/find_all` query:

Method
POST

Request URL
http://p11.flashphoner.com:9091/rest-api/stream/find_all

SEND

Parameters

Headers | Body | Variables

Body content type
application/json

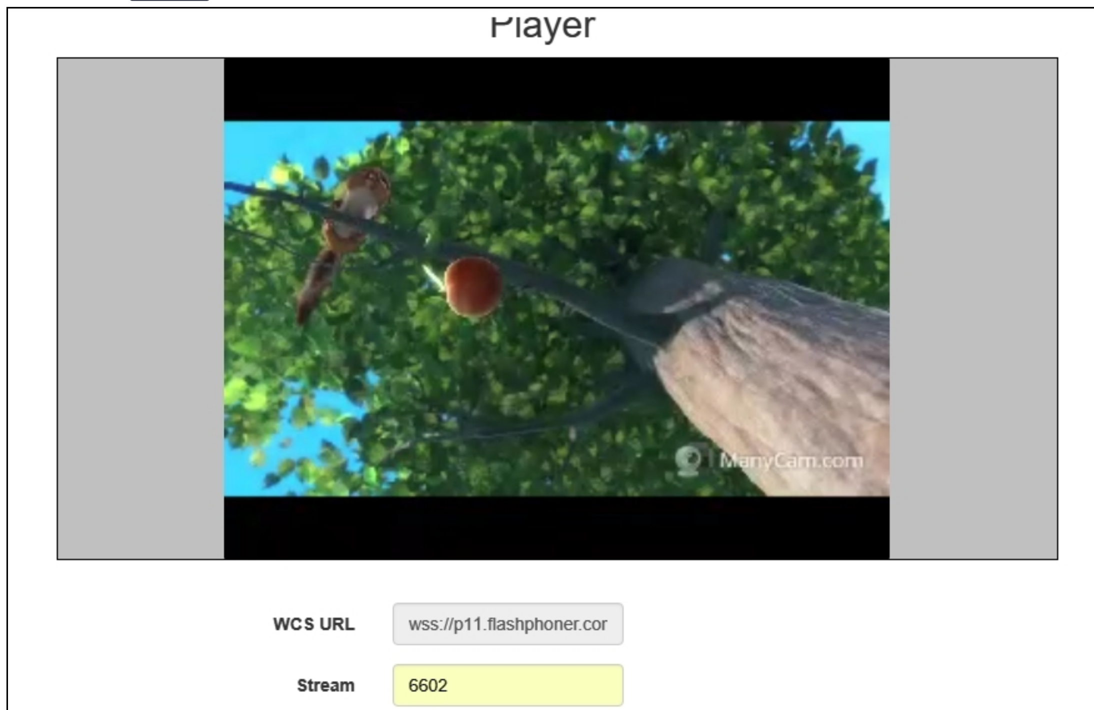
Editor view
Raw input

FORMAT JSON MINIFY JSON

200 OK 93.10 ms DETAILS

```
[Array[1]
  -0:  {
      "localMediaSessionId": "da157e2b-2159-40c9-9560-325bbe068769",
      "remoteMediaSessionId": null,
      "localStreamName": "6602",
      "remoteStreamName": "6602",
      "uri": "wss://p12.flashphoner.com:8443/websocket",
      "status": "NEW"
  }
],
```

12. Open the Player web application and put in the local stream name into the `Stream` field, then click `Start`:



Call flow

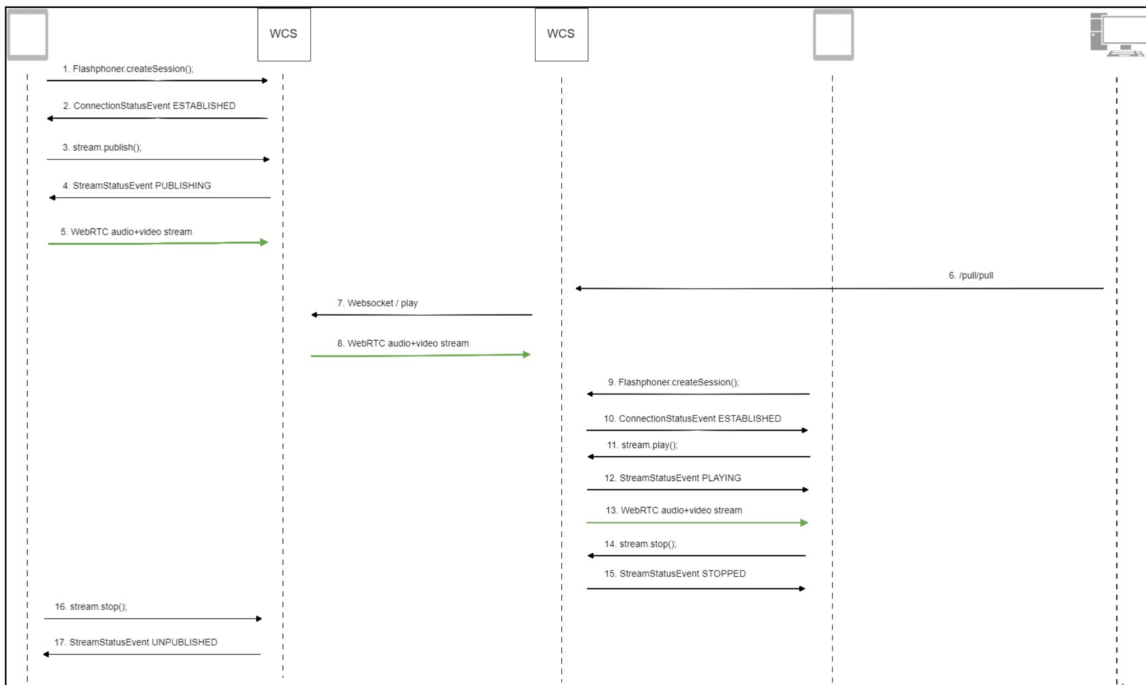
Below is the call flow when using the Two Way Streaming example to publish a stream on one WCS server and playing that stream on another WCS server

two_way_streaming.html

two_way_streaming.js

player.html

player.js



1. Establishing connection to the server

   `Flashphoner.createSession()` code

```
Flashphoner.createSession({urlServer: url}).on(SESSION_STATUS.ESTABLISHED,
function (session) {
  setStatus("#connectStatus", session.status());
  onConnected(session);
}).on(SESSION_STATUS.DISCONNECTED, function () {
  setStatus("#connectStatus", SESSION_STATUS.DISCONNECTED);
  onDisconnected();
}).on(SESSION_STATUS.FAILED, function () {
  setStatus("#connectStatus", SESSION_STATUS.FAILED);
  onDisconnected();
});
```

2. Receiving from the server an event confirming successful connection

   `SESSION_STATUS.ESTABLISHED` code

```
Flashphoner.createSession({urlServer: url}).on(SESSION_STATUS.ESTABLISHED,
function (session) {
  setStatus("#connectStatus", session.status());
  onConnected(session);
}).on(SESSION_STATUS.DISCONNECTED, function () {
  ...
}).on(SESSION_STATUS.FAILED, function () {
  ...
});
```

3. Publishing the stream

   `Stream.publish()` code

```
session.createStream({
  name: streamName,
  display: localVideo,
  cacheLocalResources: true,
  receiveVideo: false,
  receiveAudio: false
}).on(STREAM_STATUS.PUBLISHING, function (stream) {
  ...
}).on(STREAM_STATUS.UNPUBLISHED, function () {
  ...
}).on(STREAM_STATUS.FAILED, function () {
  ...
}).publish();
```

4. Receiving an event confirming successful publishing of the stream `STREAM_STATUS.PUBLISHING` code

```
session.createStream({
  name: streamName,
  display: localVideo,
  cacheLocalResources: true,
  receiveVideo: false,
  receiveAudio: false
}).on(STREAM_STATUS.PUBLISHING, function (stream) {
  setStatus("#publishStatus", STREAM_STATUS.PUBLISHING);
  onPublishing(stream);
}).on(STREAM_STATUS.UNPUBLISHED, function () {
  ...
}).on(STREAM_STATUS.FAILED, function () {
  ...
}).publish();
```

5. Sending the stream via WebRTC to the server
6. Sending the `/pull/pull` REST query to the second server
7. Requesting the stream from the first server
8. Sending the stream via WebRTC to the second server
9. Establishing connection to the second server `Flashphoner.createSession()` code

```
Flashphoner.createSession({urlServer: url}).on(SESSION_STATUS.ESTABLISHED,
function(session){
  setStatus(session.status());
  //session connected, start playback
  playStream(session);
}).on(SESSION_STATUS.DISCONNECTED, function(){
  setStatus(SESSION_STATUS.DISCONNECTED);
  onStopped();
}).on(SESSION_STATUS.FAILED, function(){
  setStatus(SESSION_STATUS.FAILED);
  onStopped();
});
```

10. Receiving from the server and event confirming successful connection `SESSION_STATUS.ESTABLISHED` code

```
Flashphoner.createSession({urlServer: url}).on(SESSION_STATUS.ESTABLISHED,
function(session){
    setStatus(session.status());
    //session connected, start playback
    playStream(session);
}).on(SESSION_STATUS.DISCONNECTED, function(){
    ...
}).on(SESSION_STATUS.FAILED, function(){
    ...
});
```

11. Requesting to play the stream `Stream.play()` code

```
stream = session.createStream(options).on(STREAM_STATUS.PENDING,
function(stream) {
    var video = document.getElementById(stream.id());
    if (!video.hasListeners) {
        video.hasListeners = true;
        video.addEventListener('playing', function () {
            $("#preloader").hide();
        });
        video.addEventListener('resize', function (event) {
            var streamResolution = stream.videoResolution();
            if (Object.keys(streamResolution).length === 0) {
                resizeVideo(event.target);
            } else {
                // Change aspect ratio to prevent video stretching
                var ratio = streamResolution.width /
streamResolution.height;
                var newHeight = Math.floor(options.playWidth / ratio);
                resizeVideo(event.target, options.playWidth, newHeight);
            }
        });
    }
    ...
});
stream.play();
```

12. Receiving an event confirming successful capturing and playing of the stream `STREAM_STATUS.PLAYING` code

```
stream = session.createStream(options).on(STREAM_STATUS.PENDING,
function(stream) {
    ...
}).on(STREAM_STATUS.PLAYING, function(stream) {
    $("#preloader").show();
    setStatus(stream.status());
    onStarted(stream);
}).on(STREAM_STATUS.STOPPED, function() {
    ...
```

```
}).on(STREAM_STATUS.FAILED, function(stream) {
    ...
}).on(STREAM_STATUS.NOT_ENOUGH_BANDWIDTH, function(stream){
    ...
});
stream.play();
```

13. Sending the stream via WebRTC to the playing client
14. Stopping playback of the stream
    `Stream.stop()` code

```
function onStarted(stream) {
    $("#playBtn").text("Stop").off('click').click(function(){
        $(this).prop('disabled', true);
        stream.stop();
    }).prop('disabled', false);
    ...
}
```

15. Receiving an event confirming successful unpublishing of the stream
    `STREAM_STATUS.STOPPED` code

```
stream = session.createStream(options).on(STREAM_STATUS.PENDING,
function(stream) {
    ...
}).on(STREAM_STATUS.PLAYING, function(stream) {
    ...
}).on(STREAM_STATUS.STOPPED, function() {
    setStatus(STREAM_STATUS.STOPPED);
    onStopped();
}).on(STREAM_STATUS.FAILED, function(stream) {
    ...
}).on(STREAM_STATUS.NOT_ENOUGH_BANDWIDTH, function(stream){
    ...
});
stream.play();
```

16. Stopping publishing the stream
    `Stream.stop()` code

```
function onPublishing(stream) {
    $("#publishBtn").text("Stop").off('click').click(function () {
        $(this).prop('disabled', true);
        stream.stop();
    }).prop('disabled', false);
    $("#publishInfo").text("");
}
```

17. Receiving an event confirming successful unpublishing of the stream
    `STREAM_STATUS.UNPUBLISHED` code

```
session.createStream({
    name: streamName,
    display: localVideo,
    cacheLocalResources: true,
    receiveVideo: false,
    receiveAudio: false
}).on(STREAM_STATUS.PUBLISHING, function (stream) {
    ...
}).on(STREAM_STATUS.UNPUBLISHED, function () {
    setStatus("#publishStatus", STREAM_STATUS.UNPUBLISHED);
    onUnpublished();
}).on(STREAM_STATUS.FAILED, function () {
    ...
}).publish();
```