Remote Worker Server Documentation

Release 1.6.0

Mozilla Services — Da French Team

April 15, 2015

Contents

Remote Worker Server is a service that handles signaling for the Firefox Remote Worker WebRTC data channel setup.

- Online documentation
- Issue tracker
- Contributing

CHAPTER 1

Table of content

1.1 Rationale

The remote worker server is a signaling service helping client-side user agents (local workers) to connect to gecko instances running in the cloud (remote workers).

The goal of the remote worker server is to connect clients with remotely-running gecko instances, in order to execute javascript workers code in a distant way.

On the gecko side, the code will be ran in a service worker.

1.1.1 Philosophy

The message broker doesn't handle the message passing between the Firefox Client and the Firefox headless server, but does the preliminary work (WebRTC data channel setup) to make this happen. Once the connection is setup, gecko talks directly with the clients using a WebRTC peer connection.

On startup, all geckos registers on the server and wait for clients to connect.

On the client side, local workers send an URL that should be run remotely, as well as a Firefox Accounts authentication information.

One Firefox Account will always be tied to one remote gecko instance.

As soon as the WebRTC data channel is up, the remote worker server connection is closed and the following exchanges are made P2P using the WebRTC data channel.

1.2 Installation

By default, Remote Worker Server uses Redis for both cache, task queuing and message passing.

1.2.1 Distribute & Pip

Installing Remote Worker Server with pip in a python3 environment:

```
pip install remote-worker-server
```

1.2.2 Install Redis

Linux

On debian / ubuntu based systems:

```
apt-get install redis-server
```

or:

```
yum install redis
```

OS X

Assuming brew is installed, Redis installation becomes:

```
brew install redis
```

To restart it (Bug after configuration update):

```
brew services restart redis
```

1.3 Getting started

Once storage engines and python dependencies have been installed, it's is easy to get started!

1.3.1 Run locally

By default, remote-worker-server persists its records inside a Redis database, so it has to be installed first (see the "Install Redis" section below for more on this).

You will also need to have a Python 3.4 running.

The server

```
make serve
```

Add a fake worker

```
make mock_worker
```

Connect a fake client

```
make mock_client
```

1.3.2 Run tests

```
make tests
```

1.4 API specifications

1.4.1 Authentication

The authentication is using *Firefox Accounts*, to verify the *OAuth2 bearer tokens* on a remote server

OAuth Bearer token

Use the OAuth token with this header:

```
{
    "authorization": "Bearer <oauth_token>"
}
```

notes If the token is not valid, this will result in a `INVALID_TOKEN` error response.

Firefox Account

Obtain the token

XXX: Endpoint still to be done.

The `GET /v1/fxa-oauth/params` endpoint can be used to get the configuration in order to trade the Firefox Account BrowserID with a Bearer Token. See Firefox Account documentation about this behavior.

```
$ http GET http://localhost:8000/v0/fxa-oauth/params -v

GET /v0/fxa-oauth/params HTTP/1.1
Accept: */*
Accept-Encoding: gzip, deflate
Host: localhost:8000
User-Agent: HTTPie/0.8.0


HTTP/1.1 200 OK
Content-Length: 103
Content-Type: application/json; charset=UTF-8
Date: Thu, 19 Feb 2015 09:28:37 GMT
Server: waitress

{
    "oauth_uri": "https://oauth-stable.dev.lcip.org",
    "scope": "remote-worker-server"
}
```

1.4.2 Client endpoint

Websocket endpoint

Requires authentication

Clients connects the websocket directly on /

```
ws://localhost:8765/
```

Creating a new worker

In order to ask for a worker creation, the client need to send this first message:

```
{
    "messageType": "hello",
    "action": "client-hello",
    "authorization": "Bearer <oauth token>",
    "source": "<worker_js_url>",
    "webrtcOffer": "<sdp-offer>"
}
```

Hello fields

- **messageType**: hello
- **action**: client-hello
- **authorization**: Bearer token
- **source**: The worker JavaScript code URL
- **webrtcOffer**: The WebRTC SDP offer

Sending ICE Candidates

Once the Hello has been sent client can start to send ICE Candidates.

```
{
    "messageType": "ice",
    "action": "client-candidate",
    "candidate": {
        "candidate": "candidate:2 1 UDP 2122187007 10.252.27.213 41683 typ host",
        "sdpMid": "",
        "sdpMLineIndex": 0
    }
}
```

ICE Candidate fields

- **messageType**: ice
- **action**: client-candidate
- **candidate**: A candidate object

Candidate object fields

- **candidate**: The candidate content
- **sdpMid**: The candidate mid

- **sdpLineIndex**: The SDP Line Index

Receiving errors

In case the gecko is not available or was not able to start the worker, an error will be returned.

```
{
    "messageType": "worker-error",
    "workerId": "<worker-id>",
    "errno": "<worker error number>",
    "reason": "<error reason>"
}
```

Receiving WebRTC Answer

If everything worked, the gecko answer will be sent back:

```
{
    "messageType": "hello",
    "action": "worker-hello",
    "workerId": "<Worker ID >",
    "webrtcAnswer": "<Gecko WebRTC Answer>"
}
```

Receiving Gecko ICE Candidates

In order to setup the WebRTC data channel, you may receive the Gecko ICE Candidates.

```
{
    "messageType": "ice",
    "action": "worker-candidate",
    "candidate": {
        "candidate": "candidate:2 1 UDP 2122187007 10.252.27.213 41683 typ host",
        "sdpMid": "",
        "sdpMLineIndex": 0
    }
}
```

1.4.3 Worker endpoint

Websocket endpoint

Worker connects the websocket on the `/worker` path.

`ws://localhost:8765/worker`

Registering the gecko

The first thing to do is to register the gecko server.

```
{
    "messageType": "hello",
    "action": "worker-hello",
    "geckoId": "<gecko-id>"
}
```

Hello fields

- **messageType**: hello
- **action**: worker-hello
- **geckoId**: The unique gecko identifier (should stay the same accross restarts)

Answering new worker demands

Receiving new worker demands

As soon as the gecko is register it will start to receive worker creation demands, of the form:

```
{
    "messageType": "new-worker",
    "userId": "<Firefox Account UserID>",
    "workerId": "<Generated Worker UUID>",
    "source": "<worker JS code source URL>",
    "webrtcOffer": "<client sdp-offer>"
}
```

When receiving this kind of request, the gecko-server should try to start the worker.

Answering with errors

In case of errors, the response should looks like:

```
{
    "messageType": "worker-error",
    "workerId": "<worker-id>",
    "errno": "<worker error number>",
    "reason": "<error reason>"
}
```

Answering with WebRTC Answer

If the Gecko was able to start the worker, it should answer with:

```
{
    "messageType": "worker-created",
    "workerId": "<worker-id>",
    "webrtcAnswer": "<gecko WebRTC answer>"
}
```

Sending ICE Candidates

Once the Answer has been sent, gecko can start to send ICE candidates.

```
{
    "messageType": "ice",
    "action": "worker-candidate",
    "candidate": {
        "candidate": "candidate:2 1 UDP 2122187007 10.252.27.213 41683 typ host",
        "sdpMid": "",
        "sdpMLineIndex": 0
    }
}
```

ICE Candidate fields

- **messageType**: ice
- **action**: worker-candidate
- **candidate**: A candidate object

Candidate object fields

- **candidate**: The candidate content
- **sdpMid**: The candidate mid
- **sdpLineIndex**: The SDP Line Index

Sending the connected status

Gecko is responsible to close the connection as soon as the WebRTC data channel is up.

This is done sending this final standza:

```
{
    "messageType": "connected",
    "workerId": "<worker id>"
}
```

1.5 Errors

1.6 Glossary

WebRTC A protocol to enable P2P connection in the browser (For audio, video and data)

Firefox Accounts Account account system run by Mozilla (https://accounts.firefox.com).

ICE Candidates A potential IP that can be use to let the other party reach us.

1.7 Contributing

1.7.1 Run tests

```
make tests
```

Run a single test

For Test-Driven Development, it is a possible to run a single test case, in order to speed-up the execution:

```
nosetests -s remote_server.tests.functional_tests:ClientServerTestCase.test_when_gecko_answers_an_of
```

1.7.2 Definition of done

- Tests pass;
- Code added comes with tests;
- Documentation is up to date.

1.8 CHANGELOG

This document describes changes between each past release.

1.8.1 0.1.0 (2015-04-15)

- Add handler for Gecko Headless websockets.
- Add handler for Client websockets.
- Handle Gecko reconnection.
- Handle Client WebRTC Offer and ICE Candidate.
- Handle Gecko WebRTC messages.
- Add documentation.

1.9 Contributors

- Fabrice Desré <fabrice@mozilla.com>
- Rémy Hubscher <rhubscher@mozilla.com>
- Mathieu Leplatre <mathieu@mozilla.com>
- Alexis Metaireau <alexis@mozilla.com>
- Tarek Ziade <tarek@mozilla.com>

CHAPTER 2

Indices and tables

- *genindex*
- *modindex*
- *search*