Seamless Offloading of Web App Computations From Mobile Device to Edge Clouds via HTML5 Web Worker Migration

Hyuk Jin Jeong
Seoul National University
SoCC 2019
Computation Offloading

Mobile clients have limited hardware resources
- Require **computation offloading** to servers
- E.g., cloud gaming or cloud ML services for mobile

Traditional cloud servers are located far from clients
- Suffer from high latency

60~70 ms (RTT from our lab to the closest Google Cloud DC)
Latency<50 ms is preferred for time-critical games

End device

Cloud data center

[Kjetil Raaen, NIK 2014]
Edge Cloud

Edge servers are located at the edge of the network

- Provide ultra low (~a few ms) latency

What if a user moves?
A Major Issue: User Mobility

How to seamlessly provide a service when a user moves to a different server?

- Resume the service at the new server
- What if execution state (e.g., game data) remains on the previous server?

This is a challenging problem

- Edge computing community has struggled to solve it
  - VM Handoff [Ha et al. SEC' 17], Container Migration [Lele Ma et al. SEC’ 17], Serverless Edge Computing [Claudio Cicconetti et al. PerCom’ 19]

We propose a new approach for web apps based on app migration techniques
Outline

Motivation

Proposed system

WebAssembly migration

Evaluation
MOTIVATION
Background: Web Apps

Apps running on a web browser
- Widely used in mobile devices due to portability
  - E.g., WebView in Android and iOS, Tizen, LG WebOS
- Program logics are written in JavaScript or WebAssembly (wasm)
  - wasm: low-level instructions for web

Web app threads
- Main thread: User interaction
- Web worker: Long-running jobs
Example: Physics Engine App

Web app simulating 3D cubes falling from the air

- **Main Thread**
  - Initialize
  - Updated locations

- **Web Worker**
  - Calculate next locations

Cube locations

<table>
<thead>
<tr>
<th>ID</th>
<th>x</th>
<th>y</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>13.4</td>
<td>44.1</td>
<td>99.1</td>
</tr>
<tr>
<td>2</td>
<td>52.6</td>
<td>79.5</td>
<td>10.5</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Example: Physics Engine App

We ran the app on the server and the client and measured FPS

- Client: Odroid XU4 (ARM CPU 2.0 GHz, 2GB memory)
- Server: Desktop PC (x86 CPU 3.6 GHz, 16 GB memory)

![Graph showing FPS with number of cubes](image)

**Observations**

1. Wasm is faster than JS (20~30%)
2. Even with wasm, client-only is not enough when # of cubes ≥ 500
Example: Physics Engine App

Web app simulating 3D cubes falling from the air

Display

Main Thread

Web Worker

Calculate next locations

Updated locations

Initialize

Cube locations (state)

<table>
<thead>
<tr>
<th>ID</th>
<th>x</th>
<th>y</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>13.4</td>
<td>44.1</td>
<td>99.1</td>
</tr>
<tr>
<td>2</td>
<td>52.6</td>
<td>79.5</td>
<td>10.5</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Computation-intensive

→ Do this on the server
Motivation: Mobile Scenario

How to continue service at the new edge server by seamlessly migrating previous edge’s state?
Previous Approach (1): VM Handoff [Ha et al. SEC 2017]

Issue
Live VM migration is heavy (due to a large base system)
- ~8 sec to migrate a Node.js instance
Previous Approach (2): Serverless computing
[Cicconetti et al. PerCom 2019]

Issue
Effective only for short-lived, stateless jobs
- The worker in our physics app has state (cube locations)
Proposed Framework: **Mobile Web Worker**

We migrate a web worker across client, edge, and cloud

- Execution state is **automatically** migrated in an application level
  - No need to migrate base systems (OS or runtime) → Lightweight

**Diagram:**
- Main thread
- Web worker
- Edge server (A): Main thread
- Edge server (B)
- Fallback server (Cloud server)

**Web worker continuously serves mobile client while preserving execution state**
Mobile Web Worker (MWW) manager controls migration of web workers and message passing with main thread

- Directly captures and restores the web worker state
  - No VM-encapsulated black box

![Diagram of Mobile Web Worker System](image)

- Worker migration
- Message channel

Client
- Web App
  - Main Thread
  - Mobile Worker

Mobile Web Worker (MWW) Manager
- Web Platform (Browser)
- OS

Edge Server
- MWW Pool
  - Mobile Worker
- Mobile Web Worker (MWW) Manager
- Web Platform (Node.js)
- OS

Cloud
- MWW Pool
  - Mobile Worker
- Mobile Web Worker (MWW) Manager
- Web Platform (Node.js)
- OS
How to Migrate Web Worker State?

Web worker is a JS program, whose runtime state consists of:
- JS scopes (variables, JS objects, functions) + events

These can be serialized into another JS code (**snapshot**) whose execution restores app state automatically [Oh et al. VEE ‘15] [Kwon et al. WWW ‘17]
- On any device equipped with a web platform

**Web Worker State**

- **Global Scope**
  - `var` `value`
  - `simulate`
  - `...` `...`
  - "message" event

- **JS function**

**Snapshot**

```
var simulate = function(m) {
  ...
};
self.addEventListener("message", simulate);
...  
```

Global scope

Event
Issues on Web Worker Snapshot

Previous snapshot implementation does not properly migrate

1. Webassembly functions
2. Built-in objects
WEBASSEMBLY MIGRATION
Background: WebAssembly (Wasm)

Low-level instruction format for web for high performance

Wasm file is translated from high-level languages (ex: C++, Rust)
  ➤ Deployed with a web app source code
  ➤ Dynamically compiled when loaded onto the browser (or JS engine)
    • After compilation, wasm function and linear memory are created
Challenges on Wasm Migration

Wasm is difficult to serialize, because

(1) Wasm file is compiled into machine code when loaded
   ➤ Compiled machine code may not run on different architecture

(2) Wasm maintains a large memory (linear memory)
   ➤ Serious transmission and recovery overhead
Proposed Method for Wasm Migration

Send a wasm file along with the code that compiles it

Linear memory is *asynchronously* transmitted and lazily restored
EVALUATION
Evaluation Environment

**Client:** Odroid XU4 (ARM 2-core CPU 2.0 GHz and 2 GB Memory) with **chromium**

**Edge server:** PC (x86 4-core CPU 3.6 GHz and 16~32 GB Memory) with **Node.js**

**Cloud server:** Google cloud (8 vCPU 2.0 GHz and 32 GB memory) with **Node.js**

**Network:** Average internet speed of US in April 2019 (mobile network, fixed broadband)
Test Applications

1. Physics simulation (ammo.js)

2. Face detection (OpenCV.js)

3. Blur filter (web-dsp)
Web Worker Migration Time

Migrating a web worker was significantly faster than migrating a Node.js VM instance

- Web worker migration does not need migration of base system

Mobile-to-edge took a long time for migration, due to

- low mobile network speed, slow mobile device
- But, it may happen infrequently

<table>
<thead>
<tr>
<th>Migration time</th>
<th>VM migration</th>
<th>Web Worker Migration</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Node.js instance</td>
<td>physics simulation</td>
</tr>
<tr>
<td>mobile to edge</td>
<td>18.2</td>
<td>3.1</td>
</tr>
<tr>
<td>edge to edge</td>
<td>7.9</td>
<td>1.0</td>
</tr>
<tr>
<td>edge to cloud</td>
<td>7.7</td>
<td>1.5</td>
</tr>
</tbody>
</table>

Unit: Second
App Execution Performance

Offloading of wasm code significantly improved app performance

- Achieved 37 FPS in physic app
- Achieved 2.6x speedup in face app, and 1.4x in filter app
  - Low speedup in filter app is due to sending input/output images

Physics

Face detection

Blur filter

Higher is better

Lower is better
Conclusion

We proposed a lightweight, state-preserving edge computing framework for web apps

The system migrates web worker using snapshot

Experiment showed promising results in both migration time and app performance
THANK YOU
Q & A