State of Loading

2017 > 2018

KenjiBaheux@, Kinuko@ + team

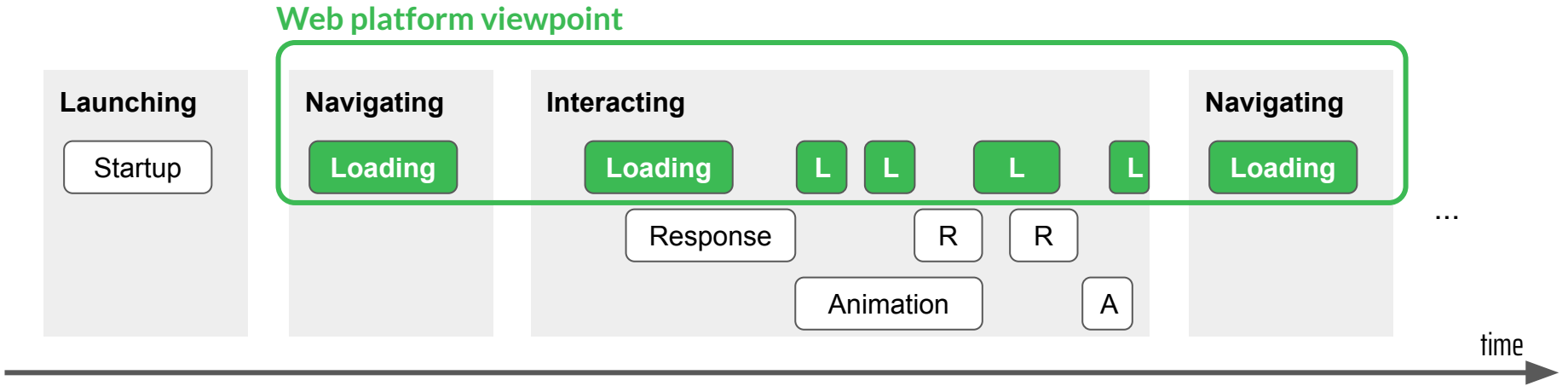
BlinkOn 8
September 20-21st 2017

Goal of This Presentation

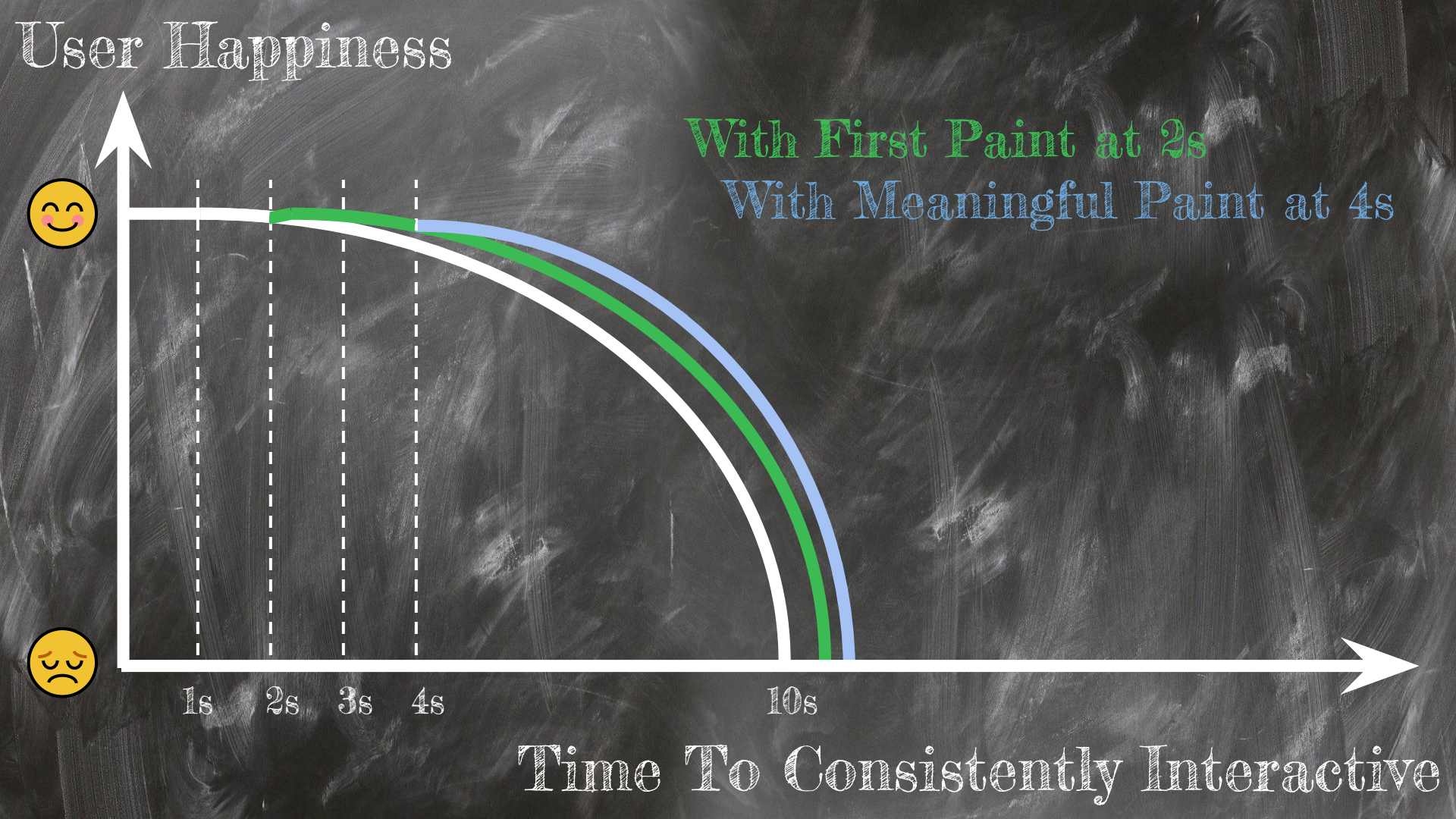
Make it righter

What is Loading?

This is Loading



When is Loading Great?


User Happiness
With First Paint at 2s
With Meaningful Paint at 4s
1s
2s
3s
4s
10s
Time To Consistently Interactive

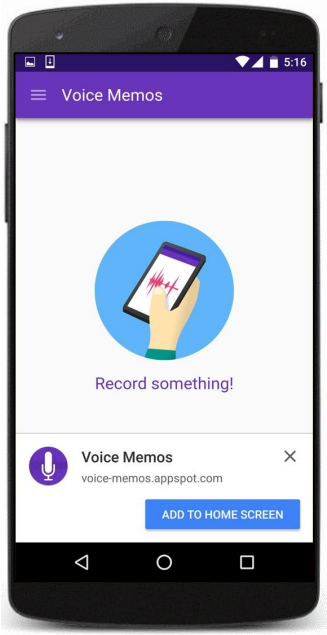

imgflip.com

Loading's Vanishing Act


VANISH
ALL THE THINGS
memegenerator.net

What to Focus on?

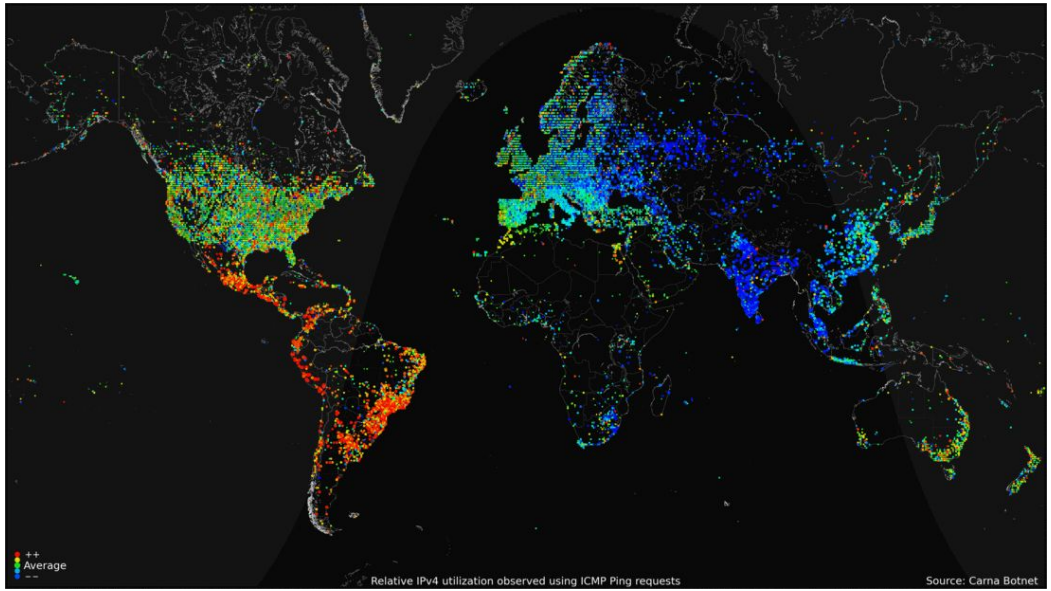
PWA & NBU



Progressive Web Apps

Naturally better user experiences

World Map of Internet Connected Devices (2012!)



Next Billion Users

Poor Loading user experience
because of slow/flaky networks

Well-Lit and Approachable Path

Problem Statement, Achievements and Plan: 2017+

Our bigger theme in 2017: Scalable Loading

Loading today does not scale!

Want to provide **delightful** loading experience for rich, modularized, progressive Web applications,

HOWEVER

What's Missing: 2017 Version

- **Developer pain**:
  - Lack of expressive primitives
  - Lack of native support for modularized Web apps
- **User pain**:
  - Critical loading path is not super-optimized
  - Non-critical loading puts janks / disturbs other work
- **Our pain**:
  - Inefficient, insecure, inflexible code with lots of debt :(

Our Focus in 2017

- **Fix Developer pain**:
  - Provide expressive primitives: Fetch, Streams etc
  - Provide native support for ES6 Modules
- **Fix User pain**:
  - Optimize critical path: PWA and Service Worker
  - Chunk work, throttle work, off-load from main thread
- **Fix Our pain**:
  - Re-architecturing!

Our Focus in 2017

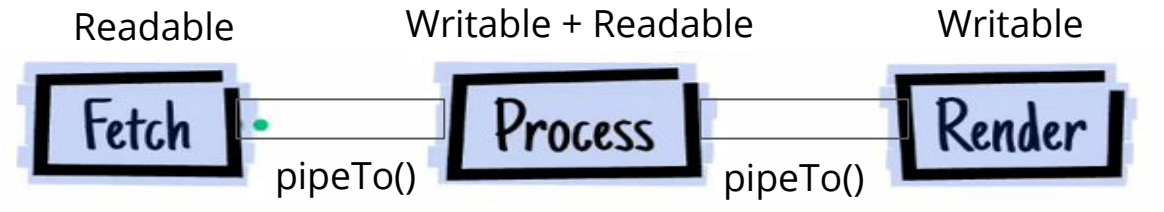
- **Fix Developer pain:**
  - Provide **expressive primitives**: Fetch, Streams etc
  - Provide **native support for ES6 Modules**
- **Fix User pain:**
  - Optimize critical path: PWA and Service Worker
  - Chunk work, throttle work, off-load from main thread
- **Fix Our pain:**
  - Re-architecturing!

Expressive Primitives

- **What we shipped / shipping (since BlinkOn 7):**
  - Streams: WritableStream and pipeTo()
  - <link rel=preload> for fetch()
  - CSS font-display
  - NetInfo API extension for Network Quality
  - fetch() with keepalive (Ongoing work)

Streams API: How streams composition could work

- Less latency, peak memory, jank
  with streaming read/write and backpressure propagation




```
async function fetchDirectlyIntoDOM() {
  const response = await fetch('catnames', ...);
  await response.body
      .pipeThrough(new TextDecoder())
      .pipeTo(targetDiv.writable);
}
```

This is just an example. Not yet Implemented

Expressive Primitives: What's coming Next?

- **What is being discussed:**
  - Priority Hints (aka "Fetch priority": JS API, HTML attributes)
  - <lazyload> to lazily load frames/images
  - Batching requests (=> less battery usage)

We can't work on many at once / want to work on what's really needed, so your input is really appreciated!

ES6 Modules

- **What we shipped / shipping (since BlinkOn 7):**
  - <script type=module> ← Yes, finally! 🎉
  - <script nomodule>
  - Modules support for Worklet

ES6 Modules: What's coming Next?

- Optimized graph fetching algorithm → **will get faster!**
- <link rel=modulepreload>
- Dynamic import()
- Import.meta
- integrity="" on <script type=module>
- Modules for Workers

```
import("./optional-feature.js")
  .then((m) => m.activate());
```

Our Focus in 2017

- **Fix Developer pain:**
  - Provide expressive primitives: Fetch, Streams etc
  - Provide native support for ES6 Modules
- **Fix User pain:**
  - Optimize critical path: **PWA and Service Worker**
  - **Chunk** work, **throttle** work, **off-load** from main thread
- **Fix Our pain:**
  - Re-architecturing!

Service Worker Optimizations: Where's the cost?

Major bottlenecks:

- **Startup**: 250ms at 50%ile on Android (before optimization)
- **Main-thread contention**: 100s of ms for startup / per fetch
- **Process hops:** ~100 ms per fetch

Startup-cost hurts no-op Service Worker latency, which tends to make developers hold off

Service Worker Optimizations

- **What we shipped / shipping (since BlinkOn 7):**
  - Navigation Preload
  - Speculative startup from Omnibox
  - Off-main-thread Fetch

- **What's in-progress:**
  - Script Streaming
  - Re-architecturing for performance

Come and listen to horo@'s lightening talk for more details!

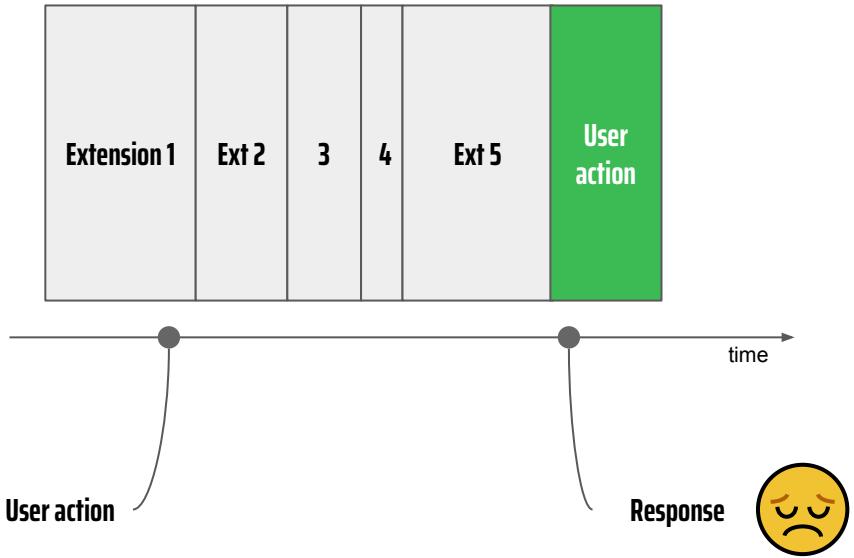
Roadmap: bit.ly/2xaybOc

Chunking and Yielding between Content Script runs

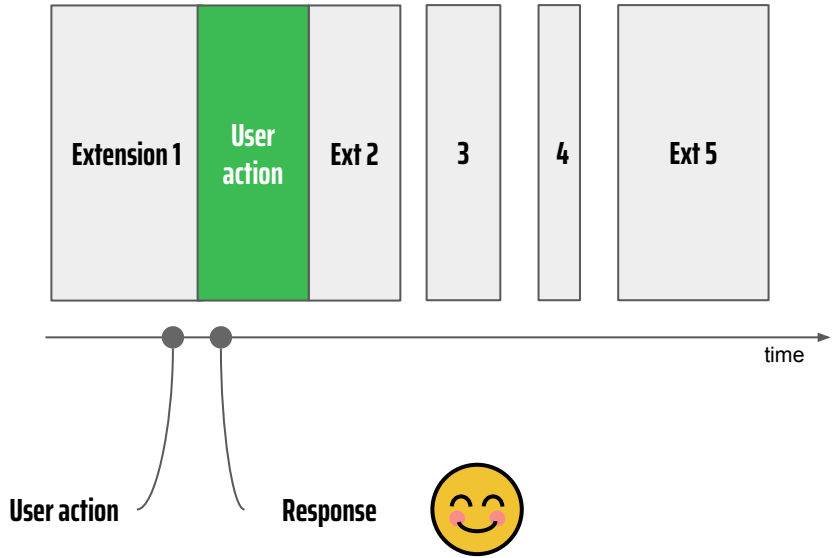
content_script for all extensions used to run in one task [**JANK**]

→ chunk it into multiple tasks asyncly!

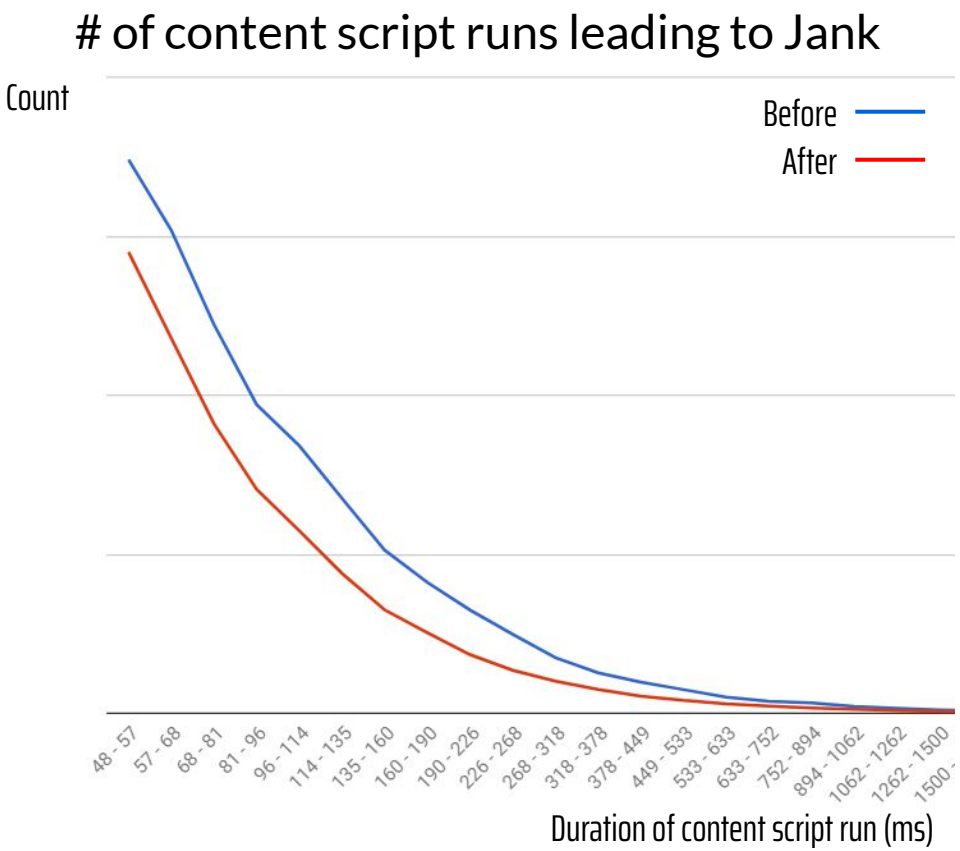
Before



After



Chunking and Yielding between Content Script runs

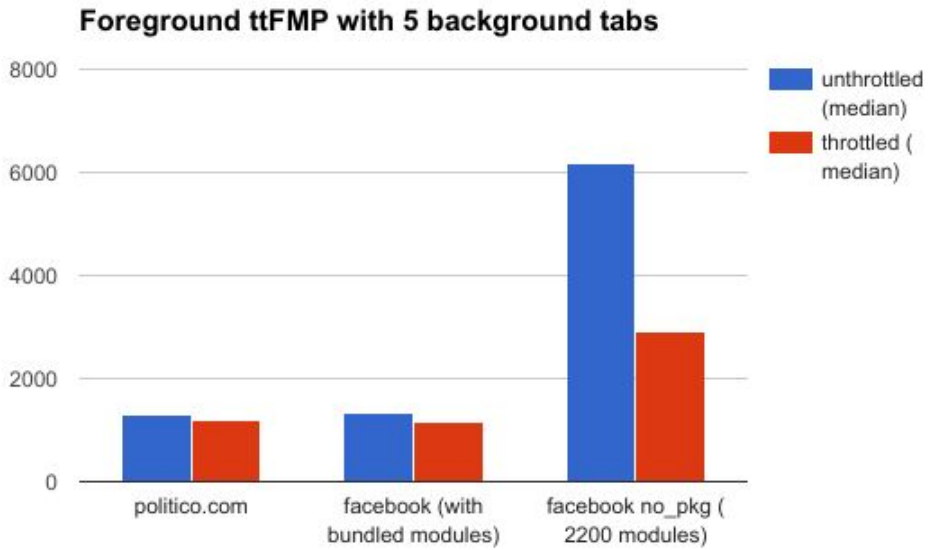


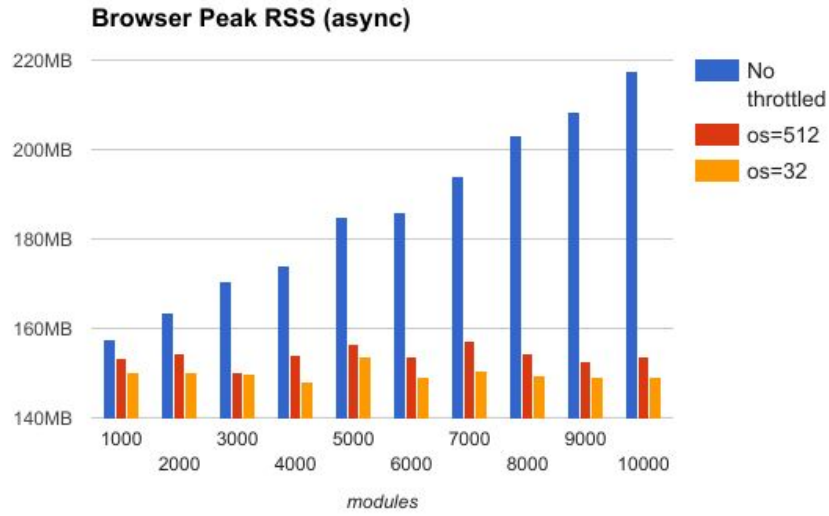
~30% less Jank caused by extensions

Canary/Dev numbers
Beta experimenting starting

Background Tab Throttling

From our in-lab local measurements (not from real world!):





Potential win for faster FMP and less OOM

if we **throttle** non-critical loading?

Background Tab Throttling

- **Throttle the loading activity in background tabs**
  - For better user experience in multi-tab scenarios
    - Less OOM, less UX jank
  - For faster ttFMP on foreground tabs

- **Status:**
  - Experimenting on Beta (waiting for more data)

Our Focus in 2017

- **Fix Developer pain**:
  - Provide expressive primitives: Fetch, Streams etc
  - Provide native support for ES6 Modules
- **Fix User pain**:
  - Optimize critical path: PWA and Service Worker
  - Chunk work, throttle work, off-load from main thread
- **Fix Our pain**:
  - **Re-architecturing!**

Re-architecturing Loading Pipeline

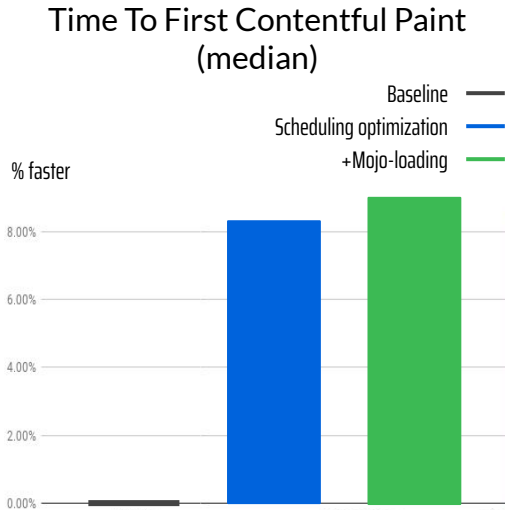
- Our Loading pipeline was designed when Chrome was born!
  - Was good, simple code initially :)

- Lots of code debt
  - All loading goes through main-thread [**JANK**]
  - Security features are added a bit ad-hocly [**CHAOS**]
  - Dumb client, assumes monolithic browser [**CAN'T S13N**]
  - A lot of legacy glue code / hooks [**NEEDS ONION SOUP**]
  - Many things work at per-process [**BAD ATTRIBUTION**]

Re-architecturing Loading Pipeline

Shipped:

- Mojo Loading (on Desktop), with **9%** TTFCP improvement :)



In-progress & Planned:

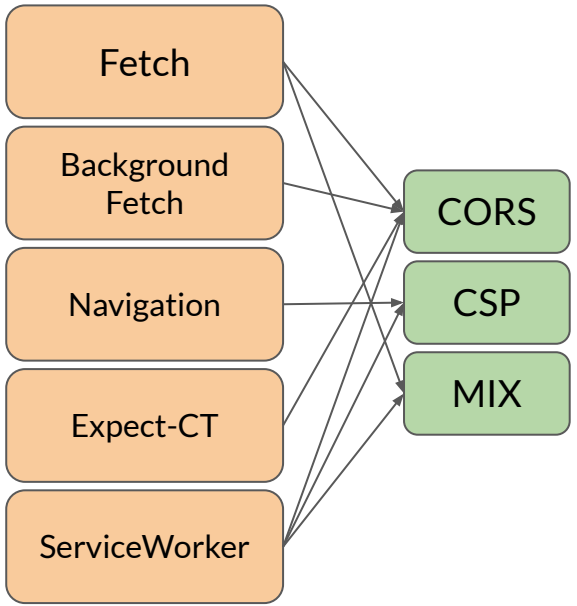
- Move CORS / CSP / MIX out of Blink
- Migrate to Per-frame loader factory
- No more RenderFrameImpl::WillSendRequest() hooks
- All loading features (Blob, AppCache, SW) use Mojo Loading
- Servicification! Onion Soup!

Unified Web Security Logic (CORS / CSP / MIX)

- Single logic shared by various components both inside/outside Blink
- Live outside of renderer for flexibility
  - sendBeacon() and keepalive fetch()

Status:

- Moving CORS out of Blink
- Cleaned up redundant var/logic of CORS



Re-architecturing Loading Pipeline

Roadmap & more details:

https://docs.google.com/document/d/10yFcHyq4ddheoJrEmyLMnCkzYvI7ioayPwH4ivDXTwg/preview

HTTP/2

Done

- Push cancellation (crbug/232040, crbug/727653).
- Proper status code when rejecting push streams (crbug/726725).

In progress

- Correct request matching for pushed responses (i.e. Vary; crbug/554220)

(re)Considering

(talk to lassey@ or kenjibaheux@)

- H2 connection reuse for non-credentials request (fetch/341).
- Cache digest: looking for evidence of value.
- Stale-while-revalidate: blocked on PlzNavigate, servicification, your input needed for prioritization!

From There...

Tentative Goals for 2018

- A well lit and approachable Loading Path.

  (e.g. Fetch Priority, Ways to take care of the hard things with Loading).

- Loading additional code/data doesn't jank the user experience.

- Efficient loading with modularized resources.

  (e.g. let you use ES6 modules as-is).

- Navigating a Multi-Page-App feels like a Single-Page-App or a Native App.

Thank You