Renderer-Side Content Decoding

horo@chromium.org (Last Update Jan 24, 2025)

crbug.com/391950057

This document is public

Introduction

This document outlines the design for "Renderer-Side Content Decoding," a performance-focused feature in Chromium that shifts the primary responsibility for content decompression from the network service to the renderer process, leveraging parallel processing within the renderer.

Prototype CL: https://chromium-review.googlesource.com/c/chromium/src/+/6169866

Goals

- Performance:
  - Reduce main IO thread contention and jank in the network service by offloading decompression to renderer processes.
  - Improve overall page load times through parallelized decompression in the renderer.
- Code Reusability:
  - Design the solution to be reusable for the Content-Addressable Disk Cache project (doc), allowing the renderer to directly access and decompress data from the cache.

Non-Goals

- Shared Dictionary Decompression: Decompression for shared dictionaries will remain in the network service. See the "Future Work" section for potential integration.
- Security Enhancements: This project is not focused on improving security. The network service will continue to handle partial decompression for ORB and MIME sniffing, and thus will still be exposed to potentially malicious compressed data.

Current Architecture

Currently, the network service is responsible for:

1. Receiving HTTP response bodies from the server, or reading HTTP response bodies from the HTTP Cache.
2. Decompressing the response bodies based on the `Content-Encoding` header.

3. Forwarding the decompressed data to the renderer process.

This can lead to performance bottlenecks, as decompression is often CPU-intensive and blocks the network service's main IO thread.

Proposed Architecture

The proposed architecture moves the main responsibility for content decompression to the renderer process, which can leverage parallel processing. The network service will retain a limited role, performing minimal decompression only for ORB and MIME type sniffing.

Key Changes

- Network Service:
  - Will perform partial decompression of the response body (up to `net::kMaxBytesToSniff` bytes) only when required for ORB and MIME type sniffing.
  - Introduces a new flag, `client_side_content_decoding_enabled`, in `network::ResourceRequest`. This flag will be `true` for requests where the renderer should handle decompression (e.g., navigation, subresource, preloading).
  - Adds a new property, `client_side_content_decoding_types` (an array of `SourceType`), to `URLResponseHead`. This array will list the content encoding types (e.g., `kGzip`, `kBrotli`, `kDeflate`) the renderer should apply, in the order specified by the `Content-Encoding` header. Importantly, if an encoding is disabled via DevTools, it will be omitted from this array even if present in the `Content-Encoding` header.
  - Sends the raw, compressed data to the renderer when `client_side_content_decoding_enabled` is `true`.
  - Error Handling: If partial decompression for sniffing fails, the network service will fail the request with `net::ERR_CONTENT_DECODING_FAILED`, similar to the current behavior.
- Renderer Process:
  - Receives the raw, compressed data from the network service if `client_side_content_decoding_types` is not empty.
  - Decompresses the data using a thread pool, applying only the decoding types listed in `client_side_content_decoding_types`.
  - Handles decompression errors gracefully.

Data Flow

1. The renderer of the browser process initiates a resource request as follows.

    - Renderer process: Subresource requests
    - Browser process: Navigation requests, preloading requests
2. Network service receives the request and fetches the response from the server.
3. Network service checks the `Content-Encoding` header and DevTools settings.
4. If `client_side_content_decoding_enabled` is `true` for this request type, and the response requires decompression:
    - If ORB or MIME type sniffing is necessary, the network service performs partial decompression (up to `net::kMaxBytesToSniff`).
    - If partial decompression fails, the network service fails the request with `net::ERR_CONTENT_DECODING_FAILED`.
    - The network service populates `client_side_content_decoding_types` in `URLResponseHead` with encoding types from `Content-Encoding`, excluding any disabled by DevTools.
    - The network service sends `URLResponseHead` (including `client_side_content_decoding_types`) and the raw, compressed data to the renderer.
5. If `client_side_content_decoding_enabled` is `false`, or the response does not need decompression:
    - The network service fully decompresses the response body (if needed) as it currently does.
    - The network service sends `URLResponseHead` and the decompressed data to the renderer.
6. The renderer process receives the `URLResponseHead` and the data.
7. If `client_side_content_decoding_types` is not empty, the renderer performs decompression based on the specified types using worker threads. Otherwise, it handles the data as received (already decompressed or no decompression needed).
    - We could potentially improve performance by decoding data when the renderer consumes the data, thus avoiding thread hops and memory copies. However, many areas of the code load data from pipes, increasing the risk of bugs where decoding is missed. Also, the error handling code will become more complex. Therefore, we'll initially implement this by adding decoding logic as soon as the renderer receives the data pipe. (prototype CL)

ORB and MIME Type Sniffing in Detail

- Opaque Response Blocking (ORB): For responses with an opaque type, the network service partially decompresses the first `net::kMaxBytesToSniff` bytes to determine if the response should be blocked based on content.
- MIME Type Sniffing: If the `Content-Type` header is missing, invalid, or ambiguous (as determined by `ShouldSniffMimeType()`), the network service partially decompresses the initial `net::kMaxBytesToSniff` bytes to infer the correct MIME type.

- Sniffing Procedure:
    1. If ORB or MIME sniffing is required, attempt to partially decompress the first `net::kMaxBytesToSniff` bytes using the encoding in `Content-Encoding` that is not disabled by DevTools.
    2. If enough data is produced for ORB/MIME sniffing, proceed with sniffing.
    3. If ORB/MIME sniffing is successful or not needed, continue to the next step.
    4. If sniffing fails due to a decompression error, fail the request with `net::ERR_CONTENT_DECODING_FAILED`.

Error Handling

- Network Service: If partial decompression for sniffing fails, the network service fails the request with `net::ERR_CONTENT_DECODING_FAILED`. No further decompression is attempted.
- Renderer Process: If any renderer decompression task fails, the request fails with `net::ERR_CONTENT_DECODING_FAILED`. The error will be passed to the network::mojom::URLLoaderClient::OnComplete() method.

DevTools Interaction

- Disabling Encodings: When a user disables a specific encoding type via DevTools, the network service will be notified and will omit that encoding type from the Accept-Encoding request header, and the `client_side_content_decoding_types` array effectively preventing the renderer from attempting to use it. This also affects the partial decompression for ORB and MIME sniffing - disabled encodings will be skipped.

Compatibility

This change maintains backward compatibility. The existing behavior (full decompression in the network service) remains the default unless `client_side_content_decoding_enabled` is explicitly set to `true` for a request.

Future Work

- Content-Addressable Disk Cache Integration: The renderer-side decompression logic will be reused for direct reading and decompression from the Content-Addressable Disk Cache (doc).
- Shared Dictionary Integration: We may consider renderer-side decompression for shared dictionary compressed response. But this requires a lot of changes in handling shared dictionary compressed responses. Currently we store the decompressed response in HTTP Cache for shared dictionary

compressed responses. We may need to change it to store the compressed response in HTTP Cache.

- Deferred Decompression: In the initial implementation, the renderer begins decoding the data in worker threads upon receiving the data pipe. However, deferring decompression until the data is actually consumed by the renderer could potentially improve performance by avoiding thread context switches and memory copies.

Metrics

- Page Load Performance: Evaluate the overall impact on page load times.