Multi-process Architecture

This document describes Chromium's high-level architecture and how it is divided among multiple process types.

Problem

It's nearly impossible to build a rendering engine that never crashes or hangs. It's also nearly impossible to build a rendering engine that is perfectly secure.

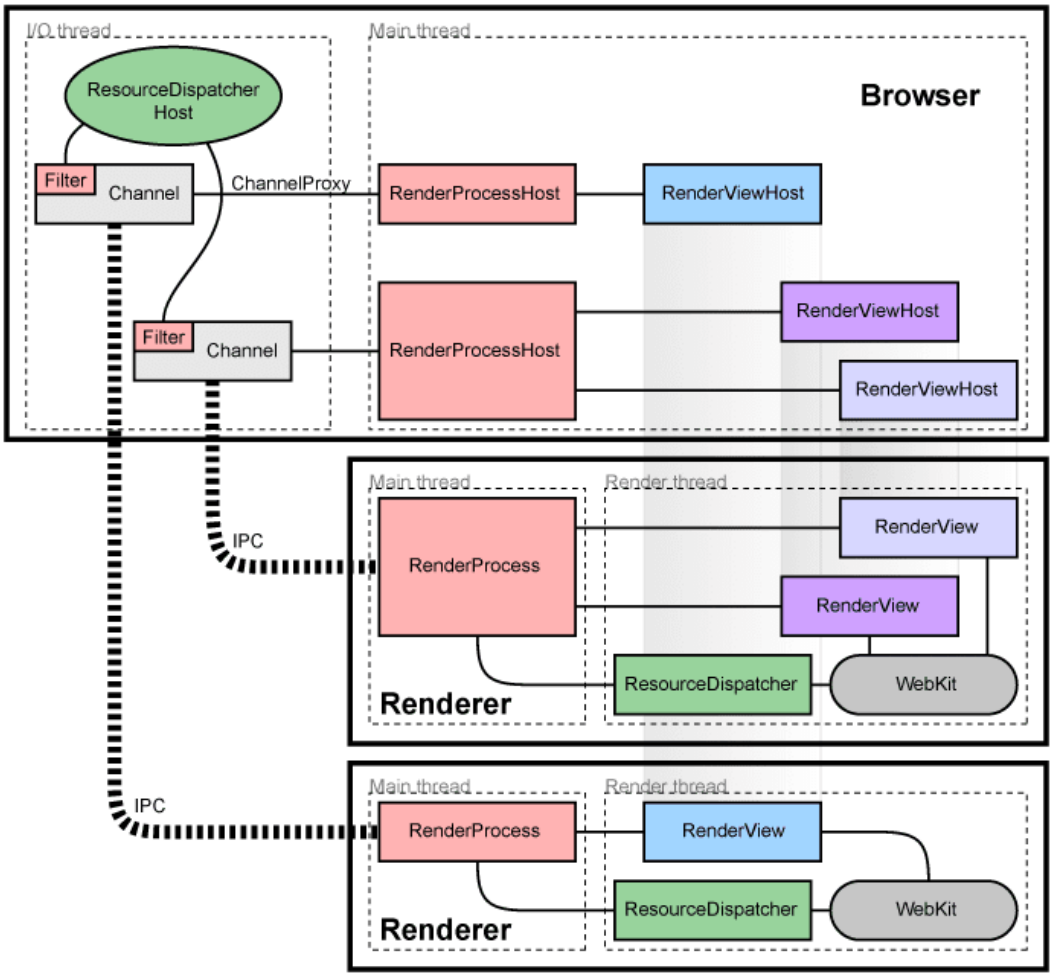
In some ways, the state of web browsers around 2006 was like that of the single-user, co-operatively multi-tasked operating systems of the past. As a misbehaving application in such an operating system could take down the entire system, so could a misbehaving web page in a web browser. All it took was one rendering engine or plug-in bug to bring down the entire browser and all of the currently running tabs.

Modern operating systems are more robust because they put applications into separate processes that are walled off from one another. A crash in one application generally does not impair other applications or the integrity of the operating system, and each user's access to other users' data is restricted. Chromium's architecture aims for this more robust design.

Architectural overview

Chromium uses multiple processes to protect the overall application from bugs and glitches in the rendering engine or other components. It also restricts access from each rendering engine process to other processes and to the rest of the system. In some ways, this brings to web browsing the benefits that memory protection and access control brought to operating systems.

We refer to the main process that runs the UI and manages renderer and other processes as the "**browser process**" or "browser." Likewise, the processes that handle web content are called "**renderer processes**" or "renderers." The renderers use the Blink open-source layout engine for interpreting and laying out HTML.



Managing renderer processes

Each renderer process has a global `RenderProcess` object that manages communication with the parent browser process and maintains global state. The browser maintains a corresponding `RenderProcessHost` for each renderer process, which manages browser state and communication for the renderer. The browser and the renderers communicate using Mojo or Chromium's legacy IPC system.

Managing frames and documents

Each renderer process has one or more `RenderFrame` objects, which correspond to frames with documents containing content. The corresponding `RenderFrameHost` in the browser process manages state associated with that document. Each `RenderFrame` is given a routing ID that is used to differentiate multiple documents or frames in the same renderer. These IDs are unique inside one renderer but not within the browser, so identifying a frame requires both a `RenderProcessHost` and a routing ID. Communication from the browser to a specific document in the renderer is done through these `RenderFrameHost` objects, which know how to send messages through Mojo or legacy IPC.

Components and interfaces

In the renderer process:

- The `RenderProcess` handles Mojo setup and legacy IPC with the corresponding `RenderProcessHost` in the browser. There is exactly one `RenderProcess` object per renderer process.
- The `RenderFrame` object communicates with its corresponding `RenderFrameHost` in the browser process (via Mojo), and the Blink layer. This object represents the contents of one web document in a tab or subframe.

In the browser process:

- The `Browser` object represents a top-level browser window.
- The `RenderProcessHost` object represents the browser side of a single browser ↔ renderer IPC connection. There is one `RenderProcessHost` in the browser process for each renderer process.
- The `RenderFrameHost` object encapsulates communication with the `RenderFrame`, and `RenderWidgetHost` handles the input and painting for `RenderWidget` in the browser.

For more detailed information on how this embedding works, see the How Chromium displays web pages design document.

Sharing the renderer process

In general, each new window or tab opens in a new process. The browser will spawn a new process and instruct it to create a single `RenderFrame`, which may create more iframes in the page (possibly in different processes).

Sometimes it is necessary or desirable to share the renderer process between tabs or windows. For example, a web application can use `window.open` to create another window, and the new document must share the same process if it belongs to the same origin. Chromium also has strategies to assign new tabs to existing processes if the total number of processes is too large. These considerations and strategies are described in Process Models.

Detecting crashed or misbehaving renderers

Each Mojo or IPC connection to a browser process watches the process handles. If these handles are signaled, the renderer process has crashed and the affected tabs and frames are notified of the crash. Chromium shows a "sad tab" or "sad frame" image that notifies the user that the renderer has crashed. The page can be reloaded by pressing the reload button or by starting a new navigation. When this happens, Chromium notices that there is no renderer process and creates a new one.

Sandboxing the renderer

Given the renderer is running in a separate process, we have the opportunity to restrict its access to system resources via sandboxing. For example, we can ensure that the renderer's only access to the network is via Chromium's network service. Likewise, we can restrict its access to the filesystem using the host operating system's built-in permissions, or its access to the user's display and input. These restrictions significantly limit what a compromised renderer process is able to accomplish.

Giving back memory

With renderers running in separate processes, it becomes straightforward to treat hidden tabs as lower priority. Normally, minimized processes on Windows have their memory automatically put into a pool of "available memory." In low-memory situations, Windows will swap this memory to disk before it swaps out higher-priority memory, helping to keep the user-visible programs more responsive. We can apply this same principle to hidden tabs. When a render process has no top-level tabs, we can release that process's "working set" size as a hint to the system to swap that memory out to disk first if necessary. Because we found that reducing the working set size also reduces tab switching performance when the user is switching between two tabs, we release this memory gradually. This means that if the user switches back to a recently used tab, that tab's memory is more likely to be paged in than less recently used tabs. Users with enough memory to run all their programs will not notice this process at all: Windows will only actually reclaim such data if it needs it, so there is no performance hit when there is ample memory.

This helps us get a more optimal memory footprint in low-memory situations. The memory associated with seldom-used background tabs can get entirely swapped out while foreground tabs' data can be entirely loaded into memory. In contrast, a single-process browser will have all tabs' data randomly distributed in its memory, and it is impossible to separate the used and unused data so cleanly, wasting both memory and performance.

Additional Process Types

Chromium has split out a number of other components into separate processes as well, sometimes in platform-specific ways. For example, it now has a separate GPU process, network service, and storage service. Sandboxed utility processes can also be used for small or risky tasks, as one way to satisfy the Rule of Two for security.