What’s Up With Mojo

With Special Guest Daniel

Today, we're talking about Mojo with Daniel. He is an IPC reviewer and has written much of the guidance and documentation around it. He’s also worked on cross-process synchronization, navigation and hardening measures to mitigate security risks.

What is Mojo

Mojo is basically Chrome's IPC system for talking between processes. [see video description for more readings]. Kind of RPC (Remote procedure call), Mojo doesn't go over the network today. Mostly used in Chrome. Legacy IPC was used before Mojo. We still have it around because NaCl (Native Client) and PPAPI (Pepper API) actually use a CIPC.

Mojo versus mojom

Mojom is specifically the file that defines your interfaces, structs, and other types that are going over Mojo IPC. Mojo is just kind of the general name for this system.

Pipes, Capabilities

All the higher-level stuff is built on top of this primitive called a message pipe. So the Mojo message pipe always has two ends. It's bidirectional. Those two endpoints can talk to each other. Capabilities are granted by interfaces to processes. For example, if the renderer has permission to the file system we would give it an interface to access the file. For a process we would only give you the interface if you have permission. And if you don't have permission, you don't have the interface at all. And you can't use the capability.

Bindings, pending, entangled, associated

Message pipes, remotes, and receivers, they come as pairs because a Mojo message pipe has two endpoints. So you always get a remote and a receiver together. **Pending** is basically the form of remotes and receivers that they are in when you can transfer them, So something has to be **pending** if you want to send it from one thread to another. Because Mojo message pipe endpoints, they're all sequence-bound, so if you want to move things between threads or between processes, they have to be in pending form. Pending means it's not handling-- it's not reading things off the message pipe or trying to send things. You can't use it in that form.

You would have to turn it from a pending into an actual remote or receiver to use it, And we have pending forms of both remotes and receivers for type safety.

Mojo message ordering is only guaranteed on the same message pipe. So if you have a remote-end receiver and you send stuff, it's a guarantee that the receiver will get things in the order you sent it in. If you call ABC, it will get ABC. But if you have two remote and receiver endpoints– if I call ABC on one and then DEF on the other, assuming they both go through the same process, there's actually no guarantee that ABC will happen before DEF so **associated** is basically a way for remotes and receivers to share an underlying message pipe.

if you have a remote, the **entangled** endpoint is the receiver on the other side or vice versa. If you have the receiver, then it's the remote on the other end.

Are there renderer-to-renderer besides renderer-to-chrome?

We have service workers, which can do interesting things with page loads, like supporting offline apps. And the way that works is you can't necessarily, from the renderer, go directly to another renderer. But the renderer, if we know it's controlled by a service worker in that document, we can give it a URL-loader factory that will actually go and talk to the service worker.
In that sense, there is renderer-to-renderer communication happening, but it's brokered. It's not just a free for all. That flexibility comes with a certain amount of danger, basically. By limiting it through a central kind of broker area, we can audit it.

Non-C++ Mojo usages.

Java. Before, people had to write a bunch of JNI boilerplate to jump from the C++ IPC handling over to Javaland. Mojo abstracts that away at some cost. There's been some persistent concerns about binary size from the Java bindings from the Android team. And they could probably be improved. There's also the JavaScript and TypeScript bindings. Chrome mostly uses the TypeScript bindings these days for things like WebUI. Some WPTs also use the JavaScript endpoints for injecting test fakes or mocks.

LaCrOS

LaCrOS is an effort to make it easier to update Chrome on ChromeOS devices.
Before, it was a monolithic thing because Chrome was also responsible for the Window environment Ash on ChromeOS. It uses versioned interfaces, which is something you won't find too much of elsewhere in Chrome, other than some ARC.

Versioned interfaces

Versioned means that these interfaces have backwards compatibility constraints because Ash Chrome and LaCrOS Chrome don't necessarily ship together. We want to be able to update LaCrOS Chrome. So we have to be able to tolerate some amount of skew between the interfaces. But we have to do it in a way that's backwards compatible. There's some complexity because of that. If you want to deprecate methods or remove fields, you can deprecate methods and remove them eventually.

Bound, connected, disconnected

Bound is when a remote or receiver isn't null. If you just default construct a Mojo remote that's not bound to-- you just default construct on, it won't be bound to anything. It'll be null internally. If you try to make a method call on it, it will crash. You actually have to create that Mojo message pipe that's backing it to, quote, unquote, "**bind**" it. So when you create that underlying Mojo message pipe, that's what it means to go from unbound to bound.

When you create a remote or receiver and you bind it, it's both bound and **connected**.
If you have a remote, you can start making method calls on it immediately. You don't have to wait for the other side to turn from pending to a receiver. Everything would just get queued. And **disconnected** is just when either endpoint is dropped. So if you drop the remote, the receiver will become disconnected. If you destroy the receiver, the remote will become disconnected. But that's an asynchronous process because it's always asynchronous, even if you're in process. But it just happens at some point. And the tricky part is if you have a bound thing, it can be disconnected. You can still make method calls on it. But your method calls will just disappear into thin air. Disconnection is a permanent thing. You can't reconnect something that was disconnected.

Sync IPC

Don’t use sync IPCs due to the potential for added complexity and performance issues. There are some cases where sync IPCs are necessary, such as older web APIs like document.cookie, but efforts have been made to optimize and reduce their usage. Additionally, there have been problems with Google integrations in Chrome due to assumptions around sync calls.