Considerations of using Mach IPC in Chromium

Author: *rsesek@chromium.org* | ***Date:*** *15 April 2015*

Several years ago, an experiment was developed to use Mach IPC to implement Chrome's IPC subsystem. At the time, there were several issues due to lack of system APIs that would provide all the necessary requirements.

Now that those issues have been resolved, there's a renewed interest in using Mach IPC. The primary motivator for this is that shared memory on Mac is slow. Switching to use Mach VM shared memory would fix this, but transferring Mach VM regions to another process requires using Mach IPC. There are two proposals for out-of-band transfer of Mach VM regions. If Chrome IPC were based on Mach IPC, then transferring a VM region would be trivial. This document *is not a formal design document*, but it discusses some of the considerations of switching Chrome's IPC layer to using Mach messaging.

IPC Channel Implementation

The base requirement is to create an IPC::Channel implementation backed by Mach messaging. Inherent in this would be providing duplex semantics over the simplex Mach channel, as Chrome IPC currently uses a duplex socketpair. This would be accomplished by having the HELLO IPC message sent by the client to the server also contain a send right to the client listening port.

The existing IPC::ChannelReader implementation also assumes stream semantics, while Mach messaging is message/record-oriented. Small refactoring here would be necessary to separate the message buffering logic from initial message cracking/routing and dispatch.

When operating in MODE_NAMED_SERVER, the Channel exports itself to the system for client connections (over a named socket or pipe). There are two possibilities for implementation:

- Each named server gets an entry in the bootstrap namespace. This could simplify the Task Port Exchange but it would also add several entries to the bootstrap namespace, and each would need to ensure it only accepts messages from legitimate communicants.
- One name would be registered in the bootstrap server, and it would demultiplex requests to the right IPC::Channel. This would require some sort of ChannelOrchestrator in the browser process.

For performance reasons, an optimal channel would use inline message transfer for small messages (<4K) and out-of-line transfer for larger messages. Before doing this, though, IPC::Channel::Send should be instrumented to determine the distribution and thresholds of message sizes. The additional complexity may not be worth the performance tradeoff for the occasional large message. (In these cases, moving the large data OOL would potentially be a better solution).

Finally, in order to pass FDs over Mach IPC, the fileport_makeport() system call will be used. This was only added in 10.6.4, so that would have to be the minimum supported Chrome version.

Message Pump

Most Chrome IPC occurs on the IO thread. This thread is backed by a libevent MessagePump. Libevent on OS X uses a kqueue to monitor FDs, but the public interface has no way to subscribe to Mach port events. In order to use use Mach messaging on the IO thread, the MessagePump must be changed. Possible options:

Modify Libevent and Pump
Libevent could be modified to support watching a Mach port (technically a port set) and reporting events on it. This would also need to be plumbed through the MessageLoopForIO interface to observe events on a port from within Chromium code.

CFRunLoop-backed IO Pump
CFRunLoop is capable of observing both FDs (as CFFileDescriptor) and Mach ports and activating callbacks for events on each. For iOS, there already exists MessagePumpIOSForIO to do this. This same code could be reused on OS X.

One risk of doing this comes from any code that relies on implicit and specific behaviors of the libevent pump. In addition, the implementation of CFRunLoop is liable to change between OS X versions.

Libdispatch IO Pump
It may be possible to run the IO thread's pump on Libdispatch. This could violate Chrome's threading assumptions by having work run serially, but on variable OS threads (whichever worker thread is available). The advantage over using CFRunLoop would be that native OS primitives (FDs and ports) wouldn't need to be wrapped in CF types that could have their own overhead and bugs.

Mach Port Exchange
With exception of the task special ports (which we shan't replace, though it is possible to do a Port Swap Dance), Mach ports are not inherited across fork(). In order for two processes to communicate, they must exchange ports. Port exchange is typically done by having the parent process register itself with the bootstrap server, the child looks up the server, and sends it port over. The Mach server registered in the bootstrap namespace would need to protect itself from arbitrary clients by using a negotiated shared secret (something akin to the existing --channel-id that isn't observable on the CLI).

While this is not conceptually difficult, doing so within Chromium's layers is. Currently, the browser process needs to gather the task port of each child process to collect process metrics. This is done by the MachBrokerMac, and children send their task ports to the browser early in startup. The task port should not be used for communication, so another Mach port exchange must occur.

One way to do this would be to provide a LaunchOptions setting to pass a series of ports to a child, just like the fds_to_remap. A class to orchestrate this exchange (via the above description) would need to exist, though how to expose this in //base is an open question. Singletons in base are generally discouraged, so a similar PortProvider/MachBroker interface and implementation split may be required.

Another option to perform the exchange would be to use MODE_NAMED_SERVER and have the child connect to it as described in the first section. This would be a divergence from the existing POSIX implementation, which passes the IPC FD over fork().