Proposal: Mojo Synchronous Methods

yzshen@chromium.org
02/02/2016

Overview

Currently there are quite a lot of sync IPC messages in Chrome: A quick search of IPC_SYNC_MESSAGE* in *messages.h returned 239 results. Some messages such as PpapiHostMsg_ResourceSyncCall are used as generic sync wrappers for other messages, so there are even more sync messages.

In order to facilitate conversion of Chrome IPC messages to Mojo interfaces, this document presents the idea of Mojo synchronous methods.

Please note: **Sync calls should be avoided whenever possible!**

- Although some sync IPCs are necessary; some might be added just for conveniences. For the latter, in-depth refactoring should be done eventually to convert them to async calls.
- Sync calls hurt parallelism and therefore hurt performance.
- Re-entrancy changes message order and produces call stacks that you probably never think about while you are coding. It has always been a huge pain.
- Sync calls may lead to deadlocks.
- As you can see below, this feature is similar to the current Chrome sync IPC support, but is less flexible (mostly intentionally). For example, with Chrome IPCs you can configure any async IPC message to “unblock” so it can re-enter ongoing sync calls at the receiving side. Such capability is not supported for Mojo sync calls. Unless later we see strong demand, I personally would like to consider lack-of-support as a feature in this case. :)

Mojom

A new attribute ***[Sync]*** (or [Sync=true]) is introduced for methods. For example:

```
interface Foo {
  [Sync]
  SomeSyncCall() => (Bar result);
};
```

It indicates that when SomeSyncCall() is called, the control flow of the calling thread is blocked[1] until the response is received.

[1] It may get re-entered. Please see the *Re-entrancy behavior* section.

It is not allowed to use this attribute with functions that don't have responses. If you just need to wait until the service side finishes processing the call, you can use an empty response parameter list:

```
[Sync]
SomeSyncCallWithNoResult() => ();
```

Message format

A new flag is defined for the flags field of message header:

```
enum {
  kMessageExpectsResponse      = 1 << 0,
  kMessageIsResponse           = 1 << 1,
  kMessageIsSync                = 1 << 2
};
```

If kMessageIsSync is set, either kMessageExpectsResponse or kMessageIsResponse must also be set.

Generated bindings (C++)

Response is mapped to output parameters. The boolean return value indicates whether the operation is successful. Returning false usually means a connection error has occurred.

```
// Client side:
virtual bool SomeSyncCall(BarPtr* result) = 0;
```

The implementation side implements a different signature:

```
// Service side:
virtual void SomeSyncCall(const SomeSyncCallCallback& callback) = 0;
```

The reason to use a signature with callback at the impl side is that the implementation may need to do some async works which the sync method's result depends on.

There are two ways to organize these signatures:
Putting them in two different interfaces:

```
class Foo {
 public:
  class Service {
    virtual void SomeSyncCall(const SomeSyncCallCallback& callback) = 0;
    ...
  };
  class Client {
    virtual bool SomeSyncCall(BarPtr* result) = 0;
    ...
  };
};
```

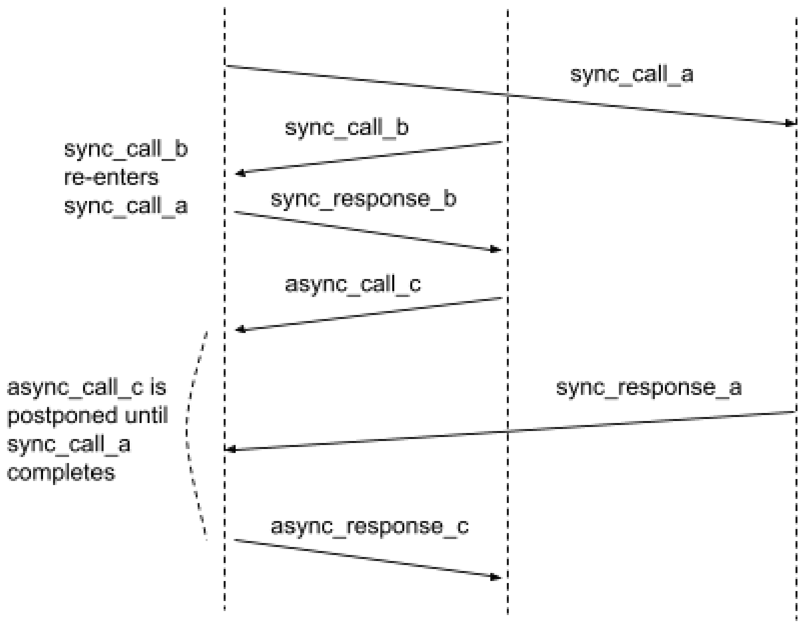
Or, put them in a single interface:

```
class Foo {
 public:
  virtual void SomeSyncCall(const SomeSyncCallCallback& callback) = 0;
  virtual bool SomeSyncCall(BarPtr* result) {
    // The service side should implement the other signature.
    NOTREACHED();
    return false;
  }
};
```

I personally prefer the second approach: it requires less changes to the existing bindings/user code; besides, it allows the client to use both the async and sync way to make the call. (That being said, the first approach is more clear about the capability/responsibility of both sides.)

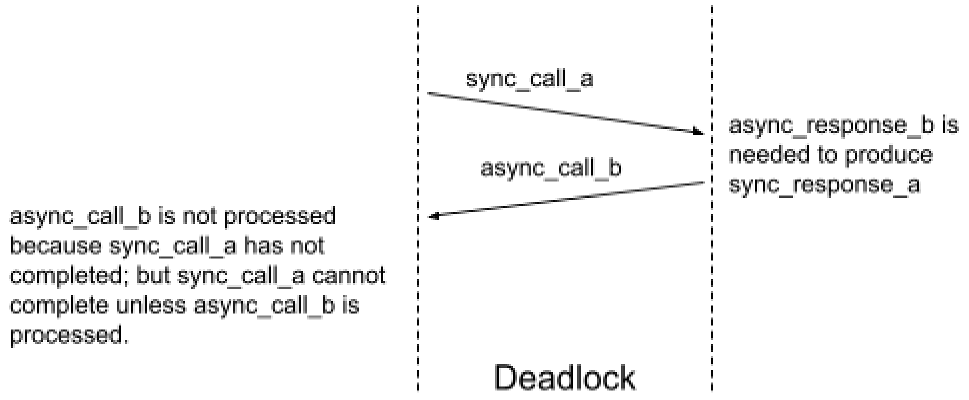
Re-entrancy behavior

What should happen on the calling thread while waiting for the response of a sync method call? This proposal adopts the following behavior: continue to process incoming sync request messages (i.e., sync method calls); block other messages, including async messages and sync response messages that don't match the ongoing sync call.


sync_call_a
sync_call_b
sync_call_b re-enters sync_call_a
sync_response_b
async_call_c
async_call_c is postponed until sync_call_a completes
sync_response_a
async_response_c

Please note that sync response messages that don't match the ongoing sync call cannot re-enter. That is because they correspond to sync calls down in the call stack. Therefore, they need to be queued and processed while the stack unwinds.

Please also note that such re-entrancy behavior doesn't eliminate deadlocks involving async calls. For example:


sync_call_a
async_response_b is needed to produce sync_response_a
async_call_b
async_call_b is not processed because sync_call_a has not completed; but sync_call_a cannot complete unless async_call_b is processed.
Deadlock

(If you find that you get into this situation, you probably want to either change async_call_b to a sync call, or use the pattern described in alternative (2) below.)

Alternatives considered (but disfavored):

1) ***no re-entrancy***: block the thread completely until the response message arrives.
Alternative (1) results in deadlocks, if two or multiple parties can issue sync calls to others and create a cycle. Besides, it is easy to achieve the same purpose using the following pattern:

```
interface SyncCallWaiter {
  Done(Bar result);
};
interface Foo {
  SyncCall(SyncCallWaiter);
};

foo->SyncCall(std::move(sync_call_waiter_ptr));
// Block and wait for the service side to call Done().
sync_call_waiter_binding.WaitForIncomingMethodCall();
```

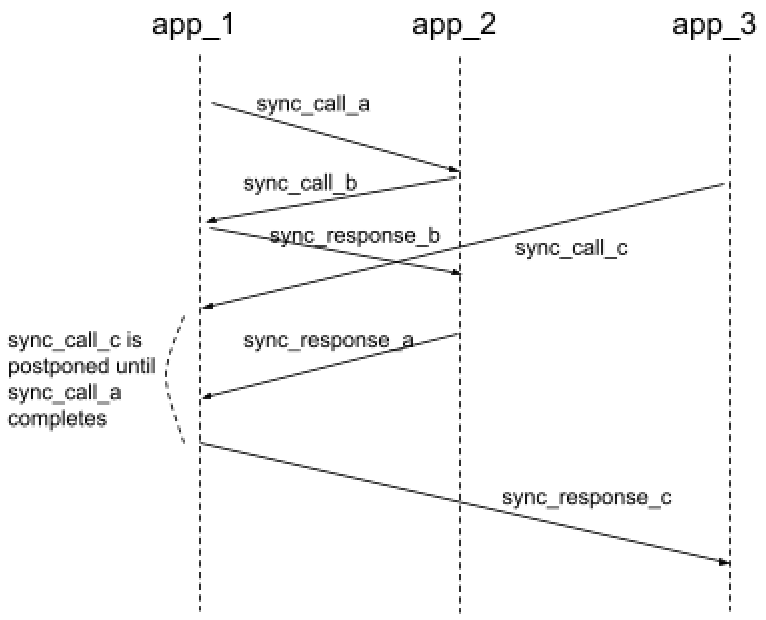
2) ***full re-entrancy***: continue to process all messages.
You can achieve the same purpose using the following pattern:

```
interface Foo {
  AsyncCall() => (Bar result);
};

foo.set_connection_error_handler([&run_loop]() { run_loop.Quit(); });
```

```
foo->AsyncCall([&run_loop](BarPtr result) { run_loop.Quit(); });
run_loop.Run();
```

3) ***context-based re-entrancy***: continue to process incoming sync request messages ***caused by*** the ongoing sync call, but not other messages. For example, in the following diagram, sync_call_b is resulted from sync_call_a so it is allowed to re-enter; on the other hand, sync_call_c is not resulted from sync_call_a so it is postponed until sync_call_a completes:



Alternative (3) seems useful to reduce sync call re-entrancy. However, it can lead to deadlocks similar to alternative (1). Also, it is hard (if not impossible) to determine automatically whether a call is “caused by” another call. Consider complicating the example above a little bit: what should we do if in order to serve sync_call_a, app_2 has to make an async call to app_4 which in turn sends sync_call_d to app_1? The bindings probably have to require the user to pass a context ID around explicitly, in order to tell whether a call is caused by another call.

Message pumping and scheduling

Basic case

Obviously, when an interface pointer (let’s call it calling_ptr) is used to make a sync call, we need to watch calling_ptr’s message pipe handle for response.

In addition, we also need to watch all bindings that serve sync calls on the same thread. A thread-local registry is necessary to keep track of all those bindings.

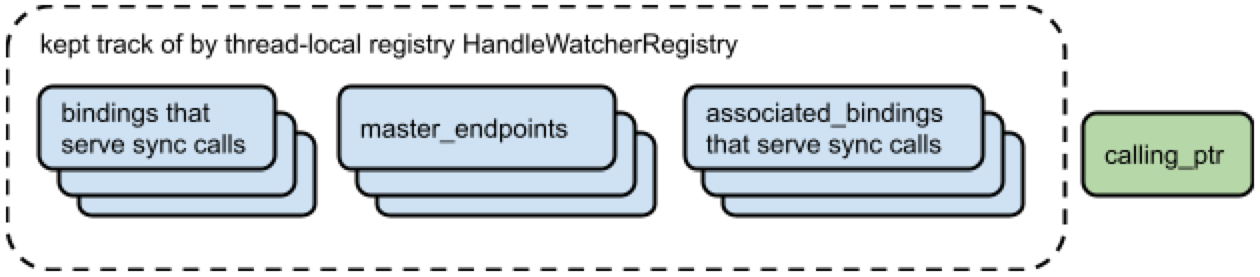
While waiting for sync response on calling_ptr, those bindings being watched may receive async requests; calling_ptr may receive async responses and non-matching sync responses (for previous sync calls down in the call stack). They all need to be queued and processed later.

More complex case: associated interfaces involved

It becomes more complex when associated interfaces are involved. Because master interface endpoints (no matter they are bindings or interface pointers) serve as routers for all associated interfaces running on the same pipe. Those associated interfaces may live on different threads. Also, associated interfaces may contain sync calls. It means we also need to watch:

- All interface ptrs and bindings that serve as master endpoints. Even if there are only async messages, the destination associated endpoints may live on a different thread. It is undesirable to block them.
- Associated bindings that serve sync calls. Because associated bindings don't own a message pipe handle, we need to set up a control message pipe between an associated binding and its corresponding master endpoint to signal about sync message arrival.

Combine all the cases above, while waiting for a sync response on calling_ptr, we will need to watch:


kept track of by thread-local registry HandleWatcherRegistry
bindings that serve sync calls
master_endpoints
associated_bindings that serve sync calls
calling_ptr

Calling_ptr could be an associated interface pointer, too. We also need to use a control message pipe between it and its corresponding master endpoint. But the rest is the same.

In some cases, users may want to enforce that certain threads shouldn't make any sync calls. It is straightforward to set such policy at the thread-local registry and enforce it when calling_ptr queries what handles should be watched.