Inter-process Communication (IPC)

Contents

Overview

Chromium has a multi-process architecture which means that we have a lot of processes communicating with each other. Our main inter-process communication primitive is the named pipe. On Linux & OS X, we use a `socketpair()`. A named pipe is allocated for each renderer process for communication with the browser process. The pipes are used in asynchronous mode to ensure that neither end is blocked waiting for the other.

For advice on how to write safe IPC endpoints, please see Security Tips for IPC.

IPC in the browser

Within the browser, communication with the renderers is done in a separate I/O thread. Messages to and from the views then have to be proxied over to the main thread using a `ChannelProxy`. The advantage of this scheme is that resource requests (for web pages, etc.), which are the most common and performance critical messages, can be handled entirely on the I/O thread and not block the user interface. These are done through the use of a `ChannelProxy::MessageFilter` which is inserted into the channel by the `RenderProcessHost`. This filter runs in the I/O thread, intercepts resource

request messages, and forwards them directly to the resource dispatcher host. See Multi-process Resource Loading for more information on resource loading.

IPC in the renderer

Each renderer also has a thread that manages communication (in this case, the main thread), with the rendering and most processing happening on another thread (see the diagram in multi-process architecture). Most messages are sent from the browser to the WebKit thread through the main renderer thread and vice-versa. This extra thread is to support synchronous renderer-to-browser messages (see "Synchronous messages" below).

Messages

Types of messages

We have two primary types of messages: "routed" and "control." Control messages are handled by the class that created the pipe. Sometimes that class will allow others to received message by having a `MessageRouter` object that other listeners can register with and received "routed" messages sent with their unique (per pipe) id.

For example, when rendering, control messages are not specific to a given view and will be handled by the `RenderProcess` (renderer) or the `RenderProcessHost` (browser). Requests for resources or to modify the clipboard are not view-specific so are control messages. An example of routed messages are a message to ask a view to paint a region.

Routed messages have historically been used to get messages to a specific `RenderViewHost`. However, technically any class can receive routed messages by using `RenderProcessHost::GetNextRoutingID` and registering itself with `RenderProcessHost::AddRoute`. Currently both `RenderViewHost` and `RenderFrameHost` instances have their own routing IDs.

Independent of the message type is whether the message is sent from the browser to the renderer, or from the renderer to the browser. Messages related to a document's frame sent from the browser to the renderer are called `Frame` messages because they are being sent *to* the `RenderFrame`. Similarly, messages sent from the renderer to the browser are called `FrameHost` messages because they are being sent *to* the `RenderFrameHost`. You will notice the messages defined in frame_messages.h are two sections, one for `Frame` and one for `FrameHost` messages.

Plugins also have separate processes. Like the render messages, there are `PluginProcess` messages (sent from the browser to the plugin process) and `PluginProcessHost` messages (sent from the plugin process to the browser). These messages are all defined in `plugin_process_messages.h`. The

automation messages (for controlling the browser from the UI tests) are done in a similar manner.

The same organization applies for other groups of messages exchanged between the browser and the renderer, as for `View` and `ViewHost` labeled messages exchanged between `RenderViewHost` and `RenderView`, defined in view_messages.h.

Declaring messages

Special macros are used to declare messages. To declare a routed message from the renderer to the browser (e.g. a `FrameHost` message specific to a frame) that contains a URL and an integer as an argument, write:

```
IPC_MESSAGE_ROUTED2(FrameHostMsg_MyMessage, GURL, in
t)
```

To declare a control message from the browser to the renderer (e.g. a `Frame` message not specific to a frame) that contains no parameters, write:

```
IPC_MESSAGE_CONTROL0(FrameMsg_MyMessage)
```

Pickling values

Parameters are serialized and de-serialized to message bodies using the `ParamTraits` template. Specializations of this template are provided for most common types in `ipc_message_utils.h`. If you define your own types, you will also have to define your own `ParamTraits` specialization for it.

Sometimes, a message has too many values to be reasonably put in a message. In this case, we define a separate structure to hold the values. For example, for the `FrameMsg_Navigate` message, the `CommonNavigationParams` structure is defined in `navigation_params.h`. `frame_messages.h` defines the `ParamTraits` specializations for the structures using the `IPC_STRUCT_TRAITS` family of macros.

Sending messages

You send messages through "channels" (see below). In the browser, the `RenderProcessHost` contains the channel used to send messages from the UI thread of the browser to the renderer. The `RenderWidgetHost` (base class for `RenderViewHost`) provides a `Send` function that is used for convenience.

Messages are sent by pointer and will be deleted by the IPC layer after they are dispatched. Therefore, once you can find the appropriate `Send` function, just call it with a new message:

```
Send(new ViewMsg_StopFinding(routing_id_));
```

Notice that you must specify the routing ID in order for the message to be routed to the correct View/ViewHost on the receiving end. Both the `RenderWidgetHost` (base class for `RenderViewHost`) and the `RenderWidget` (base class for `RenderView`) have `GetRoutingID()` members that you can use.

Handling messages

Messages are handled by implementing the `IPC::Listener` interface, the most important function on which is `OnMessageReceived`. We have a variety of macros to simplify message handling in this function, which can best be illustrated by example:

```
MyClass::OnMessageReceived(const IPC::Message& messag
e) {
  IPC_BEGIN_MESSAGE_MAP(MyClass, message)
    // Will call OnMyMessage with the message. The pa
rameters of the message will be unpacked for you.
    IPC_MESSAGE_HANDLER(ViewHostMsg_MyMessage, OnMyMe
ssage)
    ...
    IPC_MESSAGE_UNHANDLED_ERROR()  // This will thro
w an exception for unhandled messages.
  IPC_END_MESSAGE_MAP()
}

// This function will be called with the parameters e
xtracted from the ViewHostMsg_MyMessage message.
MyClass::OnMyMessage(const GURL& url, int something)
{
  ...
}
```

You can also use `IPC_DEFINE_MESSAGE_MAP` to implement the function definition for you as well. In this case, do not specify a message variable name, it will declare a `OnMessageReceived` function on the given class and implement its guts.

Other macros:

- `IPC_MESSAGE_FORWARD`: This is the same as `IPC_MESSAGE_HANDLER` but you can specify your own class to send the message to, instead of sending it to the current class.

  ```
  IPC_MESSAGE_FORWARD(ViewHostMsg_MyMessage, some_
  object_pointer, SomeObject::OnMyMessage)
  ```

- `IPC_MESSAGE_HANDLER_GENERIC`: This allows you to write your own code, but you have to unpack the parameters from the message yourself:

```
IPC_MESSAGE_HANDLER_GENERIC(ViewHostMsg_MyMessag
e, printf("Hello, world, I got the message."))
```

Security considerations

Security bugs in IPC can have nasty consequences (file theft, sandbox escapes, remote code execution). Check out our security for IPC document for tips on how to avoid common pitfalls.

Channels

`IPC::Channel` (defined in `ipc/ipc_channel.h`) defines the methods for communicating across pipes. `IPC::SyncChannel` provides additional capabilities for synchronously waiting for responses to some messages (the renderer processes use this as described below in the "Synchronous messages" section, but the browser process never does).

Channels are not thread safe. We often want to send messages using a channel on another thread. For example, when the UI thread wants to send a message, it must go through the I/O thread. For this, we use a `IPC::ChannelProxy`. It has a similar API as the regular channel object, but proxies messages to another thread for sending them, and proxies messages back to the original thread when receiving them. It allows your object (typically on the UI thread) to install a `IPC::ChannelProxy::Listener` on the channel thread (typically the I/O thread) to filter out some messages from getting proxied over. We use this for resource requests and other requests that can be handled directly on the I/O thread. `RenderProcessHost` installs a `RenderMessageFilter` object that does this filtering.

Synchronous messages

Some messages should be synchronous from the renderer's perspective. This happens mostly when there is a WebKit call to us that is supposed to return something, but that we must do in the browser. Examples of this type of messages are spell-checking and getting the cookies for JavaScript. Synchronous browser-to-renderer IPC is disallowed to prevent blocking the user-interface on a potentially flaky renderer.

Danger: Do not handle any synchronous messages in the UI thread! You must handle them only in the I/O thread. Otherwise, the application might deadlock because plug-ins require synchronous painting from the UI thread, and these will be blocked when the renderer is waiting for synchronous messages from the browser.

Declaring synchronous messages

Synchronous messages are declared using the `IPC_SYNC_MESSAGE_*` macros. These macros have input and return parameters (non-synchronous

messages lack the concept of return parameters). For a control function which takes two input parameters and returns one parameter, you would append `2_1` to the macro name to get:

```
IPC_SYNC_MESSAGE_CONTROL2_1(SomeMessage,  // Message
name
                            GURL, //input_param1
                            int, //input_param2
                            std::string); //result
```

Likewise, you can also have messages that are routed to the view in which case you would replace "control" with "routed" to get `IPC_SYNC_MESSAGE_ROUTED2_1`. You can also have `0` input or return parameters. Having no return parameters is used when the renderer must wait for the browser to do something, but needs no results. We use this for certain printing and clipboard operations.

Issuing synchronous messages

When the WebKit thread issues a synchronous IPC request, the request object (derived from `IPC::SyncMessage`) is dispatched to the main thread on the renderer through a `IPC::SyncChannel` object (the same one is also used to send all asynchronous messages). The `SyncChannel` will block the calling thread when it receives a synchronous message, and will only unblock it when the reply is received.

While the WebKit thread is waiting for the synchronous reply, the main thread is still receiving messages from the browser process. These messages will be added to the queue of the WebKit thread for processing when it wakes up. When the synchronous message reply is received, the thread will be un-blocked. Note that this means that the synchronous message reply can be processed out-of-order.

Synchronous messages are sent the same way normal messages are, with output parameters being given to the constructor. For example:

```
const GURL input_param("http://www.google.com/");
std::string result;
content::RenderThread::Get()-&gt;Send(new MyMessage(i
nput_param, &result));
printf("The result is %s\\n", result.c_str());
```

Handling synchronous messages

Synchronous messages and asynchronous messages use the same `IPC_MESSAGE_HANDLER`, etc. macros for dispatching the message. The handler function for the message will have the same signature as the message constructor, and the function will simply write the output to the output parameter. For the above message you would add

```
IPC_MESSAGE_HANDLER(MyMessage, OnMyMessage)
```

to the `OnMessageReceived` function, and write:

```
void RenderProcessHost::OnMyMessage(GURL input_para
m, std::string* result) {
  *result = input_param.spec() + " is not available";
}
```

Converting message type to a message name

If you get a crash and you have the message type you can convert this to a message name. The message type will be 32-bit value, the high 16-bits are the class and the low 16-bits are the id. The class is based on the enums in ipc/ipc_message_start.h, the id is based on the line number in the file that defines the message. This means that you need to get the exact revision of Chromium in order to accurately get the message name.

Example of this in 554011 was 0x1c0098 at Chromium revision ad0950c1ac32ef02b0b0133ebac2a0fa4771cf20. That's class 0x1c which is line 40 which matches ChildProcessMsgStart. ChildProcessMsgStart messages are in content/common/child_process_messages.h and the IPC will be on line 0x98 or line 152 which is ChildProcessHostMsg_ChildHistogramData.

This technique is particularly useful if you are dealing with crashes caused by content::RenderProcessHostImpl::OnBadMessageReceived