CSE 608 Fall 2013
Project 4 Report

Vinay Krishnamurthy, Amogh Akshintala

December 5, 2013

Introduction

This was a continuation of Project 3, and we instrumented Chromium for different platforms. We mainly targeted Linux and Android platforms. Our instrumentation will not work for Chrome running on Windows machines as we have used a few Linux syscalls, such as the open() call.

The main improvement from the previous project is that we can log in events to a SQLite DB regardless of the event sources (browser/renderer process).

Here's the list of events that are logged in as part of the instrumentation:

- Navigation events logging
- DOM events logging
- UI events
- Sensor events
- Script content

Overview of the new classes added to the source tree

Here's a list of the new classes that were added to the Chromium source tree in order to log in the desired events:

- *base/instrumenter.cc* - This file defines the main instrumenter class that speaks to the database interface. This class also has a couple of helper functions such as getting PID of the current process, getting current time etc.
- *base/instrumenter_database.cc* - This file defines instrumenter_database class, which is the database interface. This class has methods to open a database connection, add a database row, create a table etc. This is a singleton class, the single instance is mainly accessed by instrumnter.cc object.
- *content/renderer/render_instrumenter.cc* - This file defined the render_instrumenter class to be used by any renderer process. The events to be logged in from the renderer process will create an instance of this class rather than instrumenter class. This will then communicate with instrumenter class object via IPC.
- *content/browser/renderer_host/instrumenter_message_filter.cc* - This file defined Instrumenter-MessageFilter class. This class filters all the IPC message that is received from render_instrumenter class. It will then forward the received message to the instrumenter object.
- *content/common/renderer_instrumenter_messages.h* - This file has all the required IPC messages macros that is used by render_instrumenter class to send messages to the browser process.

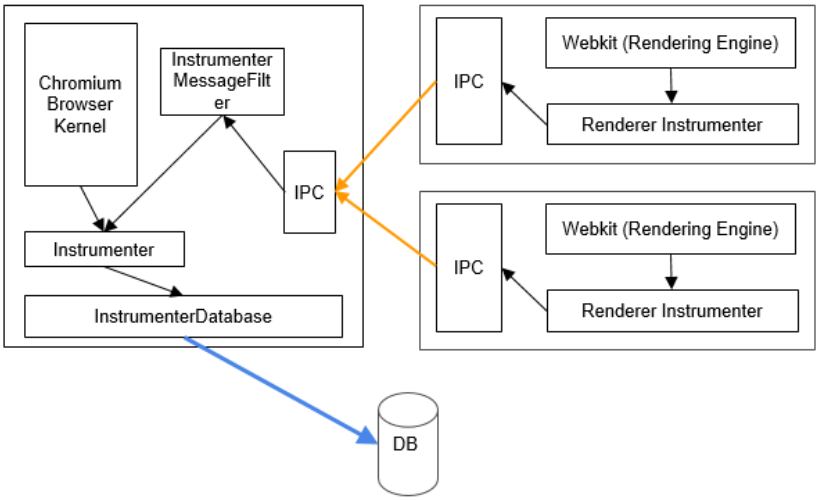

Figure 1: Interaction between various classes.

How to setup IPC channel?

As stated in the previous section, events from the renderer process are sent as IPC messages to the browser process, which in-turn calls the database interface class to write to DB. In order to setup the IPC channel, the following changes were made to the code:

1. IPC sender code was added in render_instrumenter class. We used the Send() static method of the current RenderThreadImpl class. A new routing ID is generated everytime a message is sent.
2. IPC receiver code was added in InstrumenterMessageFilter class. This involved using a bunch of predefined macros to build the IPC message map.
3. The message types defined in *content/common/renderer_instrumenter_messages.h*. We've defined three *Routing*[1] messages:
   (a) RenderMessage
   (b) RenderInitFile
   (c) RenderWriteToFile
4. We updated *content/common/content_message_generators.h* file to include the header file created in the previous step.
5. We updated the IPCMessageStart enum value in *ipc/ipc_message_start.h* to include the start value defined in header file of third step.
6. We finally added the message filter in *content/browser/render_process_host_impl.cc* file.

[1]Chrome defines 2 types of messages: *Routing* and *Control*

Where are the logs written to?

Most data is logged into an SQLite Database named InstrumentedEvents.db, that is located in the tmp directory on Linux and in the
sdcard directory on Android. The file name and the location are configurable with the *instrumenter* object constructor.

Raw data, such as scripts, is logged to individual text files. These files are named according the time they are written out and can be found in a subdirectory called File Contents in the same directory as the database.

List of changed files

Here's the list of files that we have modified to instrument the required events on the browser[2]:

1. HTTP
   (a) $ROOT/net/http/http_request_headers.cc
   (b) $ROOT/net/http/http_response_headers.cc
   (c) $ROOT/net/url_request/url_request_redirect_job.cc
   (d) $ROOT/net/url_request/url_request.cc
   (e) $ROOT/net/http/http_response_body_drainer.cc
2. GPS
   (a) $ROOT/content/browser/geolocation/location_api_adapter_android.cc
3. Accelerometer
   (a) $ROOT/content/browser/device_orientation/data_fetcher_impl_android.cc
4. Touch Screen events
   (a) $ROOT/content/browser/android/touch_point.cc
5. Media content
   (a) $ROOT/media/base/android/media_player_bridge.cc
6. Cookies
   (a) $ROOT/net/cookies/cookie_monster.cc
7. HTML content
   (a) $ROOT/third_party/WebKit/Source/core/html/HTMLFormControlElement.cpp
   (b) $ROOT/third_party/WebKit/Source/core/html/parser/HTMLSourceTracker.cpp
   (c) $ROOT/third_party/WebKit/Source/core/html/parser/HTMLSourceTracker.h
8. Script content
   (a) $ROOT/third_party/WebKit/Source/web/WebScriptController.cpp
9. Download content
   (a) $ROOT/content/browser/download/download_item_impl.h

[2]Here $ROOT = chromium/src/ directory

(b) $ROOT/content/browser/download/download_item_impl.cc

10. DOM Events
    (a) $ROOT/third_party/WebKit/Source/core/loader/FrameLoader.cpp
    (b) $ROOT/third_party/WebKit/Source/core/events/FocusEvent.cpp
    (c) $ROOT/third_party/WebKit/Source/core/events/TouchEvent.cpp
    (d) $ROOT/third_party/WebKit/Source/core/frame/DOMWindow.cpp
    (e) $ROOT/third_party/WebKit/Source/core/dom/Document.cpp
    (f) $ROOT/third_party/WebKit/Source/core/page/EventHandler.cpp
    (g) $ROOT/third_party/WebKit/Source/core/page/FocusController.cpp
    (h) $ROOT/third_party/WebKit/Source/core/events/KeyboardEvent.cpp
    (i) $ROOT/third_party/WebKit/Source/core/events/MouseEvent.cpp
11. IPC related files
    (a) $ROOT/content/common/content_message_generators.h
    (b) $ROOT/ipc/ipc_message_start.h

Automation

In order to automate the automate URL crawling with the instrumented browser, we have written 2 python scripts for each platform. The scripts take file name, which has the URL list, as an argument.

The script used for Chrome on Android is called as *run_script.py*. It uses *adb* commands to start and stop the Chrome browser activity, with the URL as an intent argument.

The Linux counterpart of this is *LinuxRun.py* that takes the following arguments:

1. path to the chrome browser
2. list of websites it should browse on Linux.

The script passes the *–disable-setuid-sandbox* to chrome as we were unable to get chrome to build successfully to use the sandbox.

We've noticed that the browser crashes for some of the sites such as `yahoo.com` and `baidu.com`. We haven't been able to find the exact root cause, but our best guess is that it could be due to overflow while sending the script content over IPC channel.