Design Doc for Plugins framework for Google Input Tools

Objective

In the IPC framework, component DLL wrapper is designed to unify the interfaces between component library and the framework, so the primary goals of the design are:

- Provide a set of unified and simple interfaces for managing the components dynamically.
- Reduce the dependency between components and the framework.
- Reduce the work of component developer.

Background

In the new cross platform input method framework, component is the basic module of functionality. And in the whole IPC framework, components can be managed dynamically, each component should be linked in a shared library. To unify the interfaces of component shared library and reduce the effort of component developer, we need to provide a dll wrapper.

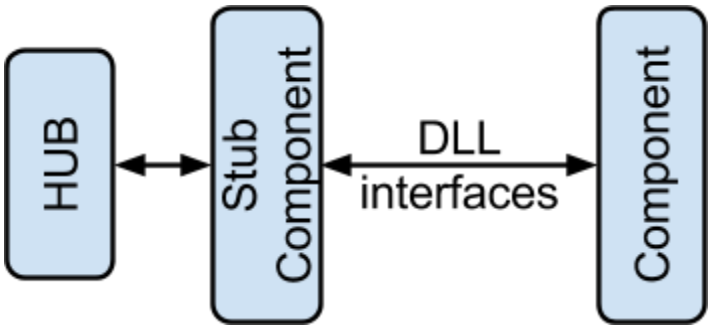
Overview

The DLL wrapper should have following functionalities:

- Provide the information of components in the dll.
- Create/Destroy a component instance.
- Delivery messages between component and IMF hub.

In the IPC framework, a component is hosted by a component host which communicates with the HUB through a message channel. The dll wrapper should as as a component host and a

message channel that can manage the component and ensure the communication between component and IPC hub:



Detailed Design

Component Informations

The component information in the dll is stored in two structures: ComponentInfo and ComponentDiscription, which is defined bellow:

```
struct ComponentInfo {
  char string_id[kComponentNameLength];
  char guid[kComponentGUIDLength];
};

struct ComponentDiscription {
  char name[kComponentNameLength];
  char author[kComponentNameLength];
  char icon_path[kPathLegth];
};
```

ComponentInfo contains string_id and guid which identify the component, and ComponentDiscription contains localized discriptions (name, author) and a optional icon which may be used by the component manager.
The component dll exports two functions (GetComponentInfos and GetComponentDiscription) to retrieve these infomation.

Component startup

All components stored in the dll wrapper should be managed within StubComponent. Creating a StubComponent instance with the parameter dll path and component id will create an instance of the actual component identified by id in the dll. The StubComponent instance will own the actual component instance.

Message delivering

The actual component communicates with the hub through the StubComponent. The StubComponent will provide the dll with a set of callback functions that allow the actual component to send message to it and it will

forward the message to the hub. The dll will exports a set of message handling functions that allow the StubComponent to forward messages to the actual component.
To make the callback functions and dll interfaces clean, we cannot pass any class type parameter in these functions in case the class definition may be changed without changing the dll. So we choose to serialize the messages into a byte buffer and pass the buffer between the StubComponent and the actual component.

Creating a component DLL

To encapsule components in a dll, you can just use the following steps:
1. Write codes for your components.
2. Defines a const ComponentInfo array named 'kComponentInfos' and stored the information of the components.
3. Defines a const int variable kComponentCount to store the number of the components in the dll.
4. Defines a CreateComponent function that creates a component instance with respect to the parameter, which is a null-terminated string id of the component.
5. link your components with dll_wrapper_lib.lib and dll_wrapper.def and creates a DLL.

Plug-in Manager Component

To manager the on disk dlls, we need a plug-in manager component to:
1. Sync the information of the plug-in components.
2. Start/Terminate a plug-in components.
3. remember the active plug-in components.
4. Unload a plug-in dll and resume all the unloaded component later.
bellow is some typical work flow of plug-in manager:

1. start up:
    when start up, the plug-in manager will:
    1. scan all the on disk dll for component informations.
    2. get the active component string ids from setttingstore.
    3. start all the active components and attach the to hub.
    4. (optional) add modification watch of plug-in directory.

2. dll change:
    when a plug-in dll changes, there's two ways of notifying the plugin manager:
    1. the installer can send MSG_UNLOAD_PLUG_IN_DLL with the dll path to the plug-in manager.
    2. The plug-in manager can watch the modification of the directory and will find the change of dll.
    when the plug-in manager is notified with the changing of plug-in dll, the plug-in manager will broadcast MSG_PLUG_IN_COMPONENTS_CHANGED with the change.

3. start/terminates a component.
    settings ui component can send MSG_START_COMPONENTS / MSG_TERMINATE_COMPONENTS to start / terminiate a set of component by there string ID.

4. Unload a plug-in dll and resumes the active components.

In the future, installer may want to update a plug-in dll without terminating the service process, so there should be a way to unload a plug-in dll before updating it and resume all the active components in the dll when update finished. Bellow is the work flow:

1. The installer sends MSG_UNLOAD_PLUG_IN_DLL to the plug-in manager component.
2. The plug-in manager will terminates all the component in the plug in dll, broadcast MSG_UNLOAD_PLUG_IN_DLL, and reply true to the MSG_UNLOAD_PLUG_IN_DLL.
3. The installer can sends the MSG_PLUG_IN_DLL_INSTALLED to the plug-in manager.
4. the plug-in manager will restart the active component in the dll when receives MSG_PLUG_IN_DLL_INSTALLED, and will broadcast MSG_UNLOAD_PLUG_IN_DLL to all the component.