A unified cross platform input method framework

last updated: 2013-03-19

Objective

The objective is to design a flexible unified input method framework that supports multilingual input and different operating systems.

The primary goals of this input method framework design are:

- Support all languages that require an input method or keyboard layout.
- Support as many as operating systems, at least: Windows, Mac and Linux
- Support different input modalities, like hardware keyboard, on-screen keyboard, handwriting recognition, voice input, character picker, etc.
- Provide a single place where the user can add, switch between, and configure input mechanisms.
- Provide the ability to "mix and match" capabilities among multiple input modalities; eg, use a virtual keyboard with a spell checker, or a voice input with a keyboard input method.

Overview

The input method framework and components

The input method framework herein is a unified infrastructure for all user text input related components in an operating system. The framework may consist of (but not limited to):

- A unified communication framework for all components to communicate with each other in a flexible and extensible way;
- A set of shared libraries and APIs to make it easier for components to utilize framework's functionalities;
- A central hub component (referred as "*IMF hub*" hereinafter) to manage running instances of all components and act like a communication and coordination bridge between components.

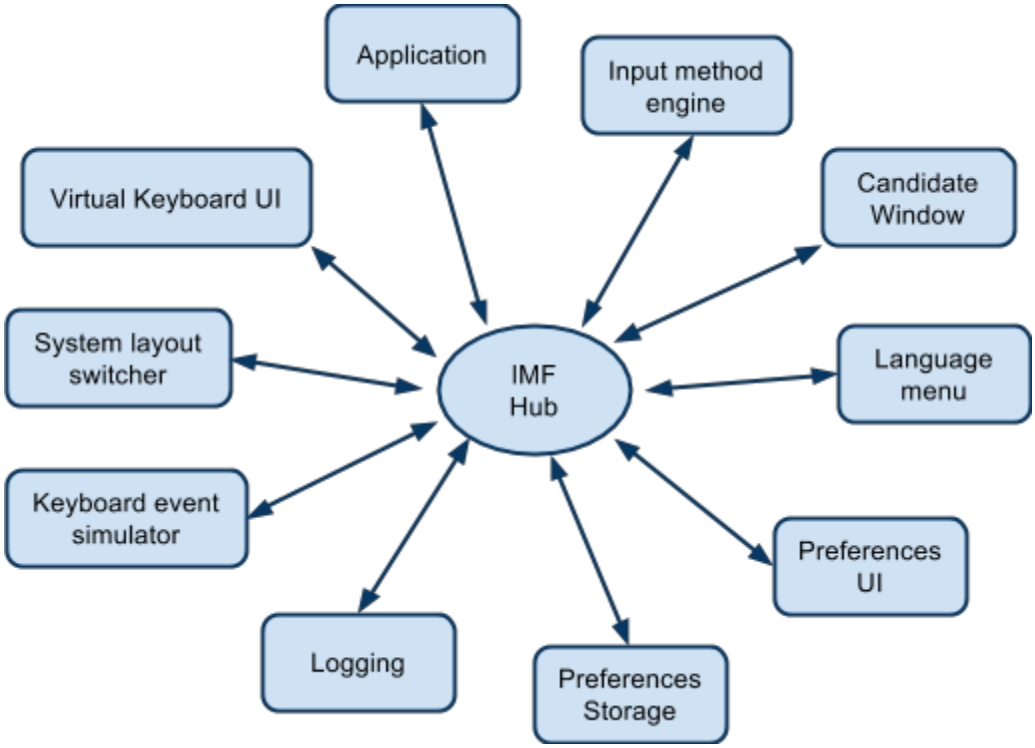
The components of this input method framework may include (but not limited to):

- Client applications.
  A client application will utilize the input method framework to support user text input with arbitrary input device and method.
- Input method engines, which take charge of converting user's input into resultant text (or information in other formats). Complex languages, such as Chinese, Japanese and Korean, have large character sets which cannot be represented by a simple keyboard, thus they require input method engines. Simple languages, such as English and European languages, may also have input method engines to provide advanced features such as spelling correction and word prediction (to reduce keystrokes).
- User interface components, such as:
  - UI for displaying auxiliary information of an input method, such as a candidates list, current status, etc.
  - UI for managing all input methods and components in the system, such as the language menu.
  - UI for non-standard input media, such as virtual keyboards, character picker and handwriting pad.
- Preferences storage component. To store preferences of all components in a unified way and optionally sync them with cloud storage.
- System related components, which provide system related functionalities to other components. Such as a component for manipulating system keyboard layouts and a component to generate simulated keyboard events (eg, for virtual keyboard).
- IMF hub itself is also a special component.

The communication framework

Reliability and robustness is a critical requirement, so each component of the input method framework may run in its own process or thread. Multiple components may run in the same process if they share the same implementation. Components connect to IMF hub via an inter-process or inter-thread communication mechanism, such as a unix domain socket, a thread local message loop, etc. Components communicate with each other by sending and receiving messages asynchronously, through IMF hub.

Below is a brief diagram of the framework:



Above diagram shows several typical types of components. It's not necessary that components with different types must be run in different processes. Some components providing multiple functionalities may run in one process. Components running in one process may share the same connection to IMF hub.

Such kind of star type architecture and message based communication ensure the flexibility and extensibility of the input method framework.

The performance overhead of this star type design would be negligible comparing to the input speed of an ordinary human. According to some benchmark results, such as: http://wiki.github.com/eishay/jvm-serializers/ and this one for dbus: http://lists.freedesktop.org/pipermail/dbus/2004-November/001779.html, which is an IPC system with similar topology.
Such star type communication framework would be able to do several thousand times of remote calls per second between two components very easily. According to the first document, Protobuf would have much higher performance comparing to dbus. Which would be far more than the input speed of an ordinary human (usually less than 5 key strokes per second).

As far as security is concerned, we shall provide a permission mechanism to declare each component's permissions for every messages. The permissions mechanism shall be enforced for all 3rd party components.

The communication protocol between components

and IMF hub

Components communicate with each other by sending and receiving messages through IMF hub. IMF hub itself can be considered as a special component.
A message consists of following information:

- A message type: describe the purpose of the message
- A reply mode flag: indicates if this message needs a reply message or itself is a reply message.
- Id of the source component: who sends the message (can also be IMF hub)
- Id of the target component: who should receive the message (can also be IMF hub)
- Id of the associated input context (for some message types)
- A serial number for identifying a message and corresponding reply message
- The payload: any data associated with the message

A message may or may not have a corresponding reply message.
Messages are classified into many groups according to their purpose, such as:

- Messages related to component registration and management
  A component must register itself to IMF hub before doing anything else. Necessary information describing the component must be provided to IMF hub. IMF hub allocates a unique id to each component, so that it can be used for further communication.
- Messages related to input context management
  An input context represents a specific application, or a specific input area in an application. A specific input method engine may be associated to an input context to provide input method service. For example, a user may be typing Japanese in one text field and English in a different text field. In a desktop session, while multiple input contexts can co-exist, only one input context can be activated (has the input focus) at a given time.
- Messages related to input events
  Such as keyboard events, mouse events, touch events, etc.
- Messages related to interactions between input method and application
  Such as manipulating composition text and surrounding context in the application, and sending result to the application.
- Messages related to input method's UI elements
  Such as candidate list, status area, etc.
- Messages related to preferences storage
- Messages related to keyboard layout and virtual keyboard support
- Customized messages between components

The type of messages can be easily extended according to requirement.

The classification of components

Different components may have different actions (or roles) for each message type. They can be classified into two types:

- Message producer (or generator)
  A component may declare that it may produce a certain type of message. For example, an application may produce a message containing a keyboard event when the user presses a key.

- Message consumer (or handler)
  A component may declare that it can handle a certain type of message. For example, an input method should be able to handle a message containing a keyboard event.

The capabilities of a component can be described by the actions of each message type supported by the component. Then components can be classified into different types according to their capabilities, such as (like above diagram shows):

- Application
- Input method engine
- Candidate Window
- Language menu
- Preferences UI
- Preferences storage
- Virtual keyboard
- Logging
- ...

The granularity of component classification is determined by the definition of message groups described above. Each type of component may have specific mandatory requirement of the actions of a minimum set of messages. For example, a component will be classified as a keyboard input method engine, only if:

- It can handle messages related keyboard input events.
- It can handle a certain type of messages related to input context operations, such as focus in and focus out.
- (Optional) It may generate messages related to composition text and result text.
- (Optional) It may generate messages related to candidate list and other UI elements.

It's not mandatory to split components with such fine-grained granularity. A component may support multiple different types of functionalities, for example, a virtual keyboard component may provide an input method engine by itself, or an integrated candidate window, that other input method engines can use. And an input method engine may also provide its own customized candidate window.

Implementations of the communication protocol, IMF hub and components

Though this design might be implemented based on many different kind of technologies, communication via protobuf would be a preferred solution.
All data types and messages will be defined with protobuf language. We may have different communication implementations (the IPC layer) based on different technologies for different environments.
For each component, the fundamental IPC interface may only consist of four methods (Names may be different in real implementation):

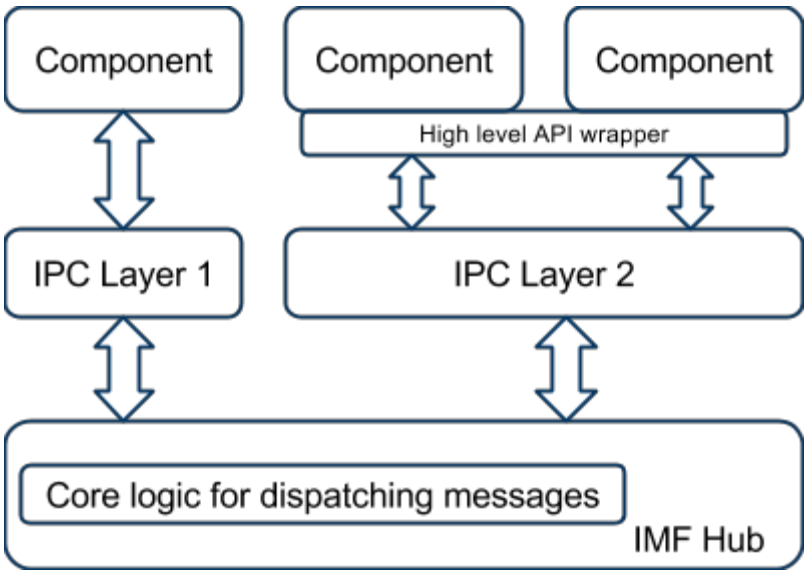
- Connect()
  Create a connection to IMF hub.
- Disconnect()

Terminate the connection.

- ReceiveMessage() (as a callback)
  Receive a message sent from another component through IMF hub.
- SendMessage()
  Send a message to another component

We need to provide a utility library to help manipulate messages. We may also provide higher level APIs and utility libraries for different type of components, to simplify the development work.

Below is a simple diagram showing the relationship between IMF hub, IPC layer and components:



This diagram shows that two IPC layers with different implementations are attached to IMF hub. Components can connect to IMF hub through an IPC layer directly or through a higher level API wrapper on top of an IPC layer.

Integration with cloud services

This design simplifies the integration with cloud services by following points:

- Component based architecture
  As we can define message types and add services by providing components freely, we can implement special components that provide cloud services to other components in this framework. So that any component can use the cloud services by sending designated message types.
- Fully asynchronous message communication framework
  This fully asynchronous message communication framework can simplify the handling of slow network access a lot. If a component wants to use a cloud service, it can just send

out the designated message without waiting for the reply. The result will be delivered back to the component asynchronously when ready.

Detailed Design

Basic concepts

Input Context

An input context is the mechanism for managing the communication between the application and the input method. It's a combination of an application window, an input method, a set of additional components used by the input method or the application and various internal state information. When opening a window for text input, the application is responsible for sending a request to IMF hub to create an input context. Upon receiving the request, IMF hub creates an internal data structure to hold the information of the input context, and depending on the information provided by the application and some other criteria, IMF hub chooses an input method component to serve the request, which may create an internal session exclusively for handling all input tasks related to the input context. Then IMF hub may select a set of additional components to serve various requirements of the application and the input method, eg. the UI requirement. IMF hub then store all of these information into the internal input context data structure and assign an id to it. Later on, any component may access the input context information by the id.
All text input related communication among the application, the input method and other additional components (eg. UI component) will be bound to a specific input context.

Input Focus

Usually in a desktop login session, at the same time, only one window can receive keyboard input, which is the so called focused window. If a window is focused and if it contains an input context, then the input context may be focused as well. In this case, all keyboard events will be delivered to the focused input context for processing and components (eg. UI component) attached to the focused input context may be adjusted to fit the position and style of the focused window.
Only one input context in a desktop login session (aka. a IMF hub instance) can get input focus at the same time. IMF hub maintains the state of the focused input context and perform necessary tasks when it's being changed.

Message

As already mentioned above, a message is the minimum data unit that can be transferred between two components. The purpose of a message is determined by its type. Some message types may require to be bound to a specific input context, such as input related messages between input methods and applications. Some other messages may have nothing to do with input context, for example messages related to preferences storage.

Component

As already mentioned above, a component is a functional block which can produce and consume a certain set of messages with different types. The type or purpose of a component can usually be classified according to message types it can consume and produce.

Relationship among Input Context, Message and Component

A component may be attached to an input context, so that it can produce or consume specific message types bound to this input context.
Multiple components, which can consume a same message type, may be attached to an input context at the same time. In this case, only one of them may be assigned as the active consumer of the message type for this input context. A component can also be attached to multiple input contexts at the same time.
Different input contexts may have different attached components assigned as the active consumer of the same message type.
This kind of relationship is very critical for IMF hub to determine the target component when dispatching a message (see below for details).

IMF hub

This section describes the detailed design of IMF hub's major functionalities.

Component management

One of the most important functionalities of IMF hub is managing all running components.

Usually components run in separated processes than IMF hub. So a certain inter-process communication mechanism may be used between component processes and IMF hub process. When a component process starts up, it creates a connection to IMF hub process, and all following communication goes through the connection. A component process may contain multiple components, which share the same connection to IMF hub. A components to connection mapping table is maintained inside IMF hub, so that it can determine the correct connection when sending a message to a specific component.
Some message types may not be bound to a specific component, such as the one for registering components to IMF hub, which should be handled by the component process itself.
Components and IMF hub may also run within different threads in a single process. In this multi-thread model, different in-process IPC mechanism may be used, such as thread-local message queue.

Component registration/deregistration

When a component process starts up, the first thing to do is to register its components into IMF hub, which is done by sending MSG_REGISTER_COMPONENT message along with ComponentInfo objects of its components to IMF hub. Upon receiving the message, IMF hub will allocate unique ids for components and send them back to the component process.
A component will be deregistered by IMF hub, if:

1. a MSG_DEREGISTER_COMPONENT message with id of the component is sent to IMF hub, or
2. the connection to the component process which owns the component is broken.

Component startup and shutdown

In order to make IMF hub small and robust, we may use a separated helper component for managing component binaries. Information of installed components may be cached on disk and components may be started on-demand.

Query component information

IMF hub may support querying the information of registered components by various criteria specified in a template ComponentInfo object.

Input context management

Input context is the most important part for tying different components together, especially the application and the input method.
Each registered component may be attached to one or more input contexts (including the default one, see below).

The default input context (kInputContextNone)

To ease the input context and component management, a default input context will be created by IMF hub upon initialization. Id of the default input context will always be kInputContextNone. All registered components will be attached to this input context by default. For messages which do not relate to any input context, kInputContextNone will be used. Then the same logic can be applied to both none-input context related and input context related messages.

The creation and deletion of an input context

Only application components may request IMF hub to create or delete an input context. The corresponding messages are MSG_CREATE_INPUT_CONTEXT and MSG_DELETE_INPUT_CONTEXT. IMF hub may use them to identify if a component is an application.
To create an input context, an application sends a MSG_CREATE_INPUT_CONTEXT message to IMF hub. The message may contain information related to the application's text input area, for example the accepted unicode scripts or character sets, the type of accepted text (text, number, date, etc.). Upon receiving the message, IMF hub will allocate a unique id for the input context and determine the necessary components (such as input method and ui component) to be attached to the input context according to the information provided by the application. For each of these components, IMF hub will send a MSG_TO_ATTACH_INPUT_CONTEXT message, so that the component can do necessary initialization for the input context.
An application owns all input contexts created by itself, and has some special privileges about these input contexts:

- An input context can only be deleted by its owner application.
- The owner application has the highest privilege for consuming messages associated to the input context, unless the application itself abandon the privilege explicitly. In another word, upon creating an input context, the owner application itself will be assigned to this input context as active consumer of all message types it can consume. No other component can take over the active consumer role from the application.
- Only the owner application can give focus to an input context.

The focused input context

Only one input context in an IMF hub instance can be focused at the same time. A special id: kInputContextFocused can be used to denote the focused input context when sending a message.

Message dispatching

When sending a message, the source component must specify the id of the target component as well as the input context id. IMF hub determines the correct target component based on these two ids:

- If the target component id is a valid component id, then the message will be sent to the target component directly regardless of the input context id, unless the target component can't consume the message type.
- If the target component id is kComponentDefault, then IMF hub will find out a suitable target component from the components attached to the specified input context.
- If only one attached component can consume the message type, then it'll be selected as the target component. Otherwise, if multiple consumer components are attached to the input context, the one currently assigned as the active consumer of the message type will be selected as the target component.
- If the input context id is kInputContextNone, then the default input context will be used in above logic. All components will be attached to the default input context automatically upon registration, the first registered consumer component of a certain message type will be automatically assigned as active consumer of this message type for the default input context.
- If the target component id is kComponentBroadcast, then the message will be dispatched to all consumer components of the message type attached to the input context.

Active consumer management

If a component is assigned as active consumer of a certain message type for the default input context, then this component will be the default active consumer of the message type.
As soon as an input context is created by an application or a component is attached to an input context, the application or component can send out a MSG_REQUEST_CONSUMER message containing a set of message types to IMF hub. Upon receiving this message, IMF hub will lookup the default active consumer components for these message types and attach them to the input context by following steps:

- Send a MSG_ATTACH_TO_INPUT_CONTEXT message to each of these default active consumer components. Meanwhile, attach them to the input context with a special pending flag indicating that they are not really attached to the input context yet.
- Upon receiving this MSG_ATTACH_TO_INPUT_CONTEXT message, the component should do necessary initialization for serving the input context and reply the message with a true Boolean value. If it fails to do initialization, false should be replied.
- If IMF hub receives a reply message with true Boolean value from a component, then it will really attach the component to the input context and assign it as active consumer of ALL message types it can consume, unless another component is already assigned as active consumer of the message type.

- If IMF hub receives a reply message with false Boolean value from a component, then it will remove the component from the pending list and find another similar component as replacement and perform the same steps for this component.
- Upon being attached to an input context, the component will receive a MSG_COMPONENT_ACTIVATED message containing all message types that it has been assigned as the active consumer for the input context.
- A component will never be attached to an input context by above steps if it can not consume MSG_ATTACH_TO_INPUT_CONTEXT.

Unlike steps listed above, a component can also be activated explicitly for an input context, either a real one or the default input context, by following steps:

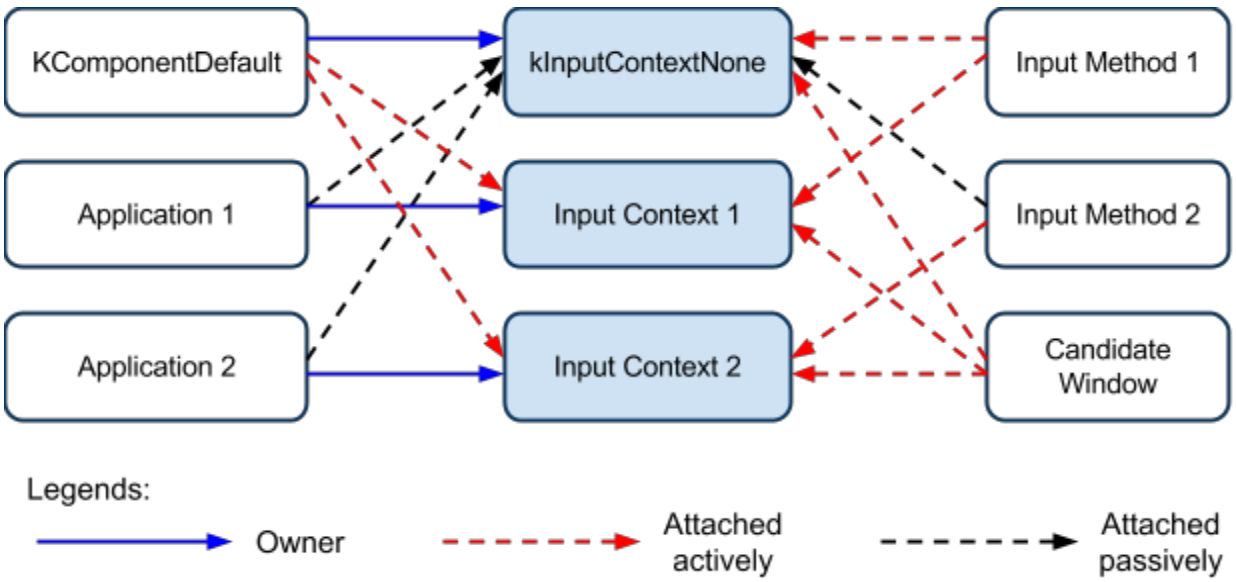
- Send a MSG_ACTIVATE_COMPONENT message containing id of the component to be activated to IMF hub. This message can be sent from a different component than the one to be activated.
- Upon receiving this message, IMF hub will send out a MSG_ATTACH_TO_INPUT_CONTEXT to the component to be activated, if the component can consume this message, then goes through the same steps listed above. Otherwise, IMF hub will attach the component to the input context directly. In this case, the component may be acknowledged by a MSG_COMPONENT_ACTIVATED message.

The difference between these two activation mechanisms are:

- In order to be attached to an input context with above steps involving MSG_REQUEST_CONSUMER message, a component must be able to consume MSG_ATTACH_TO_INPUT_CONTEXT message. While MSG_ACTIVATE_COMPONENT message can attach any component to an input context explicitly even if the component can not consume MSG_ATTACH_TO_INPUT_CONTEXT.
- MSG_REQUEST_CONSUMER message can only attach a component passively, that is, it will not replace any existing active consumers already assigned to the input context. While, MSG_ACTIVATE_COMPONENT message attaches a component actively, that is, it will assign the component as active consumers of all message types the component can consume for the input context, thus it may replace existing active consumers of those message types.

A concrete example of above component activation logic is switching among different input method components. When there are multiple registered input method components, because they can consume the same set of message types, only one of them can be set as active consumer of these message types for the default input context. This activated input method component is so called the current active input method. Then when an application creates an input context, according to above logic, this new input context will inherit the default active input method component from the default input context. Later on, if the user chooses a different input method from the language menu (a UI component), the language menu can activate the new input method component for the input context by sending a MSG_ACTIVATE_COMPONENT message. Another example is, if the active input method component wants to use a specific UI component, it can activate that specific UI component by sending a MSG_ACTIVATE_COMPONENT message as well.

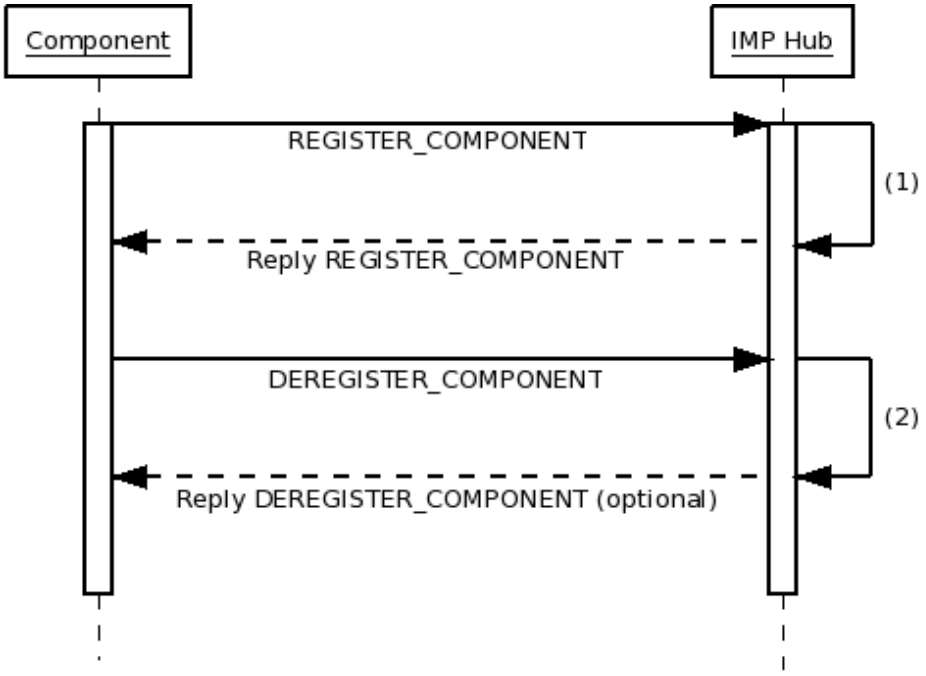
Below is a simple diagram showing the relationship among components and input contexts:



In this diagram, kComponentDefault is a special component representing IMF hub itself. kInputContextNone is the so called default input context, which is created and owned by IMF hub. Application 1 creates and owns Input Context 1, Application 2 creates and owns Input Context 2. Input Method 1 is the default active input method and also the active input method of Input Context 1. Input Method 2 is not the default active input method, but it's the active input method of Input Context 2. Candidate Window is the only registered UI component which can consume candidate list related messages, and it's attached actively to all input contexts, including the default one.

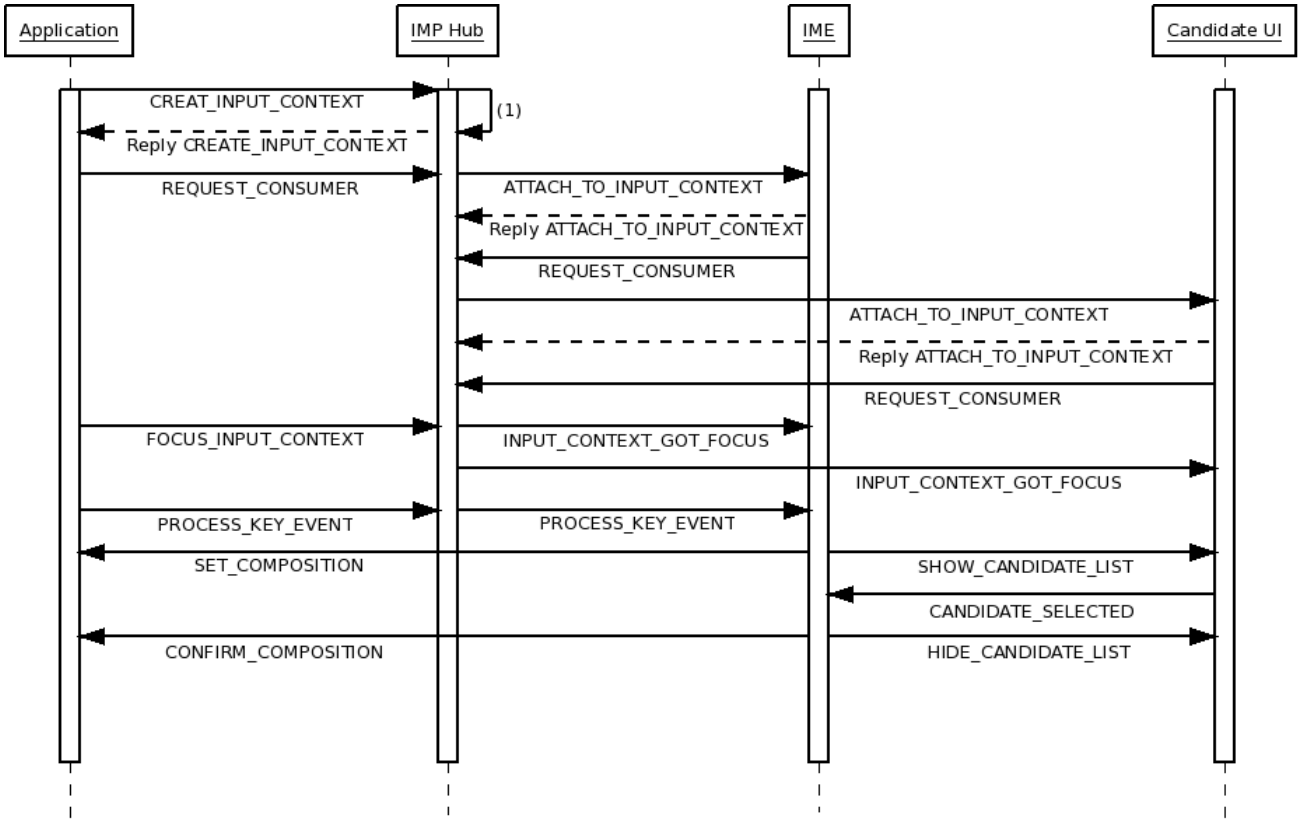
Message Sequence Diagrams

Register and deregister a component



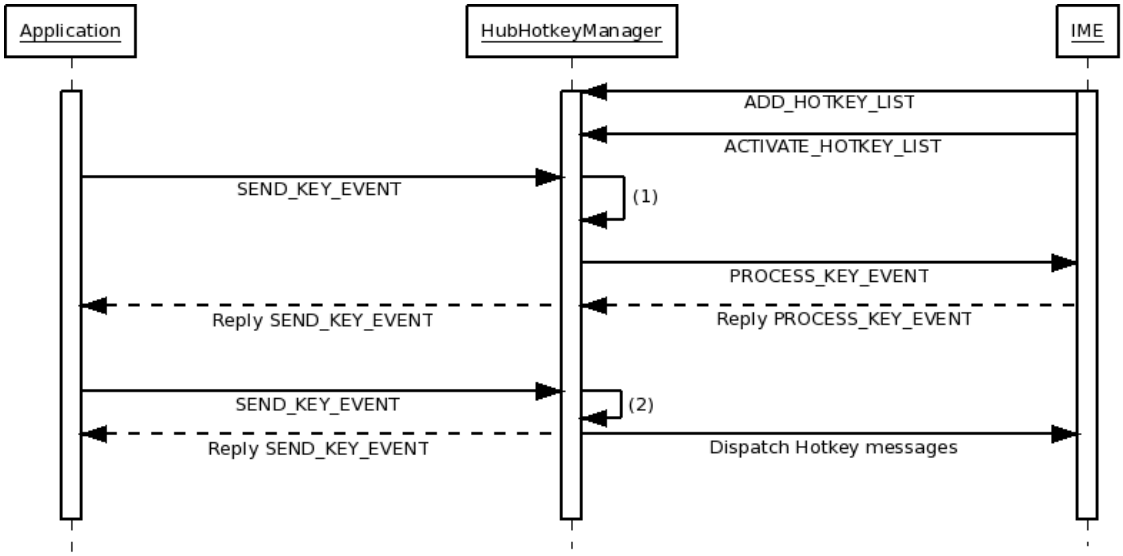
(1) Create internal Component object and broadcast COMPONENT_CREATED message when necessary.
(2) Delete internal Component object and broadcast COMPONENT_DELETED message when necessary.

Create Input Context and Basic keyboard event handling



(1) Create internal InputContext object and broadcast INPUT_CONTEXT_CREATED message when necessary.

Hotkey matching



(1) If the key event is not a Hotkey, then send the key event to the IME and put the key event into a pending list for upcoming reply event.

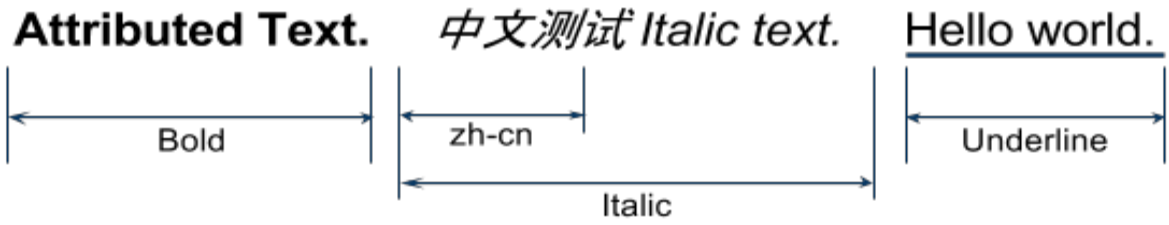
(2) If the key event is a Hotkey, then dispatch messages of the Hotkey and send back reply event to the Application immediately

Basic object types

This section describes some predefined object types. This section is just a rough introduction rather than a reference. It's not a complete list.

Attributed Text

An attributed text is a piece of multilingual text with its text attributes. An attribute spans a certain range in the text denoting a specific attribute of the text inside the range. Multiple attributes can overlap with each other. Below is an illustration of an attributed text:



The types of attribute may have:

- Rendering related attributes, such as font information, foreground and background color, underline, etc.
- Language and linguist related attributes, such as language, clause segment, ruby annotation, part-of-speech tag, etc.
- Other informations related to the text, such as composition state, hyperlink, pronunciation data, etc.

Candidate

Candidates are alternative characters for a given input sequence. A candidate object may optionally contain following members:

- An optional icon of the candidate.
- An attributed text containing the content of the candidate.
- A hint text of the candidiate, which usually can be a concise meaning of the candidate or any other simple descriptive information.
- An annotation of the candidate, which can be a more detailed description of the candidate, or other related information. It can be a piece of attributed text or HTML, or other data types types supported by the UI component.
- A CandidateList object containing multiple sub candidates. Which can be used to group several similar candidates into one top candidate entry.
- A CommandList object containing one or more commands associated to the candidate. The commands can be some operations to the candidate, such as: "Add it to user dictionary", "Delete it from user dictionary", etc. Commands may be presented as a context menu of the candidate.

CandidateList

A CandidateList object contains a list of candidates along with some additional informations controlling how to present candidates on the screen. It may optionally contain following members:

- A title and a footnote containing some descriptive information of candidates in the list.
- Page and scrolling information controlling how to draw the scrollbar and page flipping buttons.
- A set of flags controlling some behaviors of the UI, such as whether or not the list can be shrunk in horizontal or vertical direction, the layout direction of each candidate, etc.
- Page size of the candidate list, which may have three different layout forms:
    - A 1 row by multiple columns horizontal layout
    - A multiple rows by 1 column vertical layout
    - A multiple rows by multiple columns two dimension layout
- Index of the selected candidate.
- A list of unicode characters to be displayed as selection keys of each candidate.
- Candidate objects of the current page.

A candidate list can only contain candidates of the current page. Page flipping actions shall be handled by the input method itself.

Command

A command represents an action supported by a component. For example an input method engine component can support a command for changing current input mode, and maybe another one for launching a user dictionary management tool.
A component can support arbitrary number of commands. Commands can be organized in hierarchical manner. A component may register its commands to a dedicated UI component for presenting. The UI component may present commands registered by all components in a way suitable for the operating system's UI style. For example, on Mac, commands may be presented as menu items in system's language menu, while on Windows, they may be presented as buttons on system's language bar.
Usually, commands of an input method engine component are bound to specific input contexts. An input method component engine may register its commands when one of its attached input context gets focused. The UI component may only show these commands while the context is being focused. Any other components may register/deregister commands any time after being initialized. These commands may be displayed by the UI component regardless of the focused input context.
A command can be triggered by the user by any means supported by the UI component, such as mouse click or touch.
A command object may optionally contain following members:

- A unified integer key for identifying the command. It must be unified inside the component scope.
- A flag indicating the state of the command, which may be:
    - Normal, indicating it's a normal command
    - Checked, indicating its corresponding functionality is activated

    - Mixed, indicating some of its corresponding functionalities are activated
    - Separator, indicating it's just a separator between two different type of commands
- A flag indicating whether the command is enabled or not. A user can only trigger an enabled command.
- A text label corresponding to the current state
- An icon corresponding to the current state
- A tooltip
- A CommandList object containing one or more sub commands. The sub commands may be presented by the UI component in a sub menu or as indented menu items, or any other style supported by the UI component.

CommandList

A CommandList object is used for containing multiple commands.

KeyEvent

A KeyEvent object contains necessary information of a keyboard event. It may optionally contain following members:

- A type, key down or key up.
- A platform independent virtual keycode
- A platform dependent hardware scancode
- A platform dependent keycode
- A bitmask value indicating the modifiers state
- A string containing the text generated by the key event, with modifiers applied
- A string containing the text generated by the key event, without modifiers applied

Hotkey

A hotkey is a key event with one or more messages associated to it. Hotkeys are handled by IMF hub. When a key event is sent to IMF hub, before being sent to the destination component, it'll firstly be matched against current active hotkeys registered by all components. If a hotkey is matched, then IMF hub will send out the messages associated to the hotkey rather than forwarding the key event itself.

HotkeyList

A HotkeyList object is used for containing multiple hotkeys, it may contain following members:

- A unique id to identify it in the component scope.
- one or more hotkeys.

A component can register multiple HotkeyLists into IMF hub, but only one of them can be activated for an input context. The one activated for the default input context becomes a global hotkey list, which will take effect for all keyboard events.

Message

A message object is the minimum data unit transferring between two components. It may contain following members:

- A message type: describe the purpose of the message

- A reply mode flag: indicates if this message needs a reply message or itself is a reply message.
- Id of the source component: who sends the message. A special value may be used for IMF hub.
- Id of the target component: who should receive the message. Special values may be used for IMF hub, and a broadcast message.
- An input context id. It's only valid for input context related messages.
- A serial number for identifying a message and corresponding reply message
- The payload: any data associated with the message

If a message does not need reply, then the message receiver may not send back a reply message at all. Otherwise, the message receiver should always send back a reply message. The type of the payload depends on the message type and the reply mode.

ComponentInfo

A ComponentInfo object is used for containing all necessary information of a component when registering into IMF hub. It may contain following members:

- A unique string id to identify this component
- An integer id allocated by IMF hub.
- A localized name
- A localized description
- An icon
- A list of language IDs supported by the component
- A list of Unicode Script IDs supported by the component (Besides ASCII)
- A list of message types may be produced by the component
- A list of message types may be consumed by the component

Upon initialization a component shall create a ComponentInfo object and send it to IMF hub for registration. IMF hub then sends back the integer id of the component.
A component can lookup the information of other components by the value of any members listed above.

InputContextInfo

An object to hold the information of an input context. It may contain following members:

- An integer id allocated by IMF hub.
- Id of the owner component.

InputCaret

An object to hold information related to the current input caret.

References

- Windows - Text Service Framework
- Windows - Input Method Manager

- Mac OS X - Input Method Kit Framework
- Linux - ibus project