Mashup Architecture for Connecting Graphical Linux Applications Using a Software Bus

Mohamed-Ikbel Boulabiar, Gilles Coppin, Franck Poirier

Mashup Architecture for Connecting Graphical Linux Applications Using a Software Bus

Mohamed-Ikbel Boulabiar
Lab-STICC
Telecom Bretagne, France
mohamed.boulabiar@telecom-bretagne.eu

Gilles Coppin
Lab-STICC
Telecom Bretagne, France
gilles.coppin@telecom-bretagne.eu

Franck Poirier
Lab-STICC
University of Bretagne-Sud, France
franck.poirier@univ-ubs.fr

ABSTRACT

Although UNIX commands are simple, they can be combined to accomplish complex tasks by piping the output of one command, into another's input. With the invasion of GUI applications, we have lost this ability to chain many small tools in order to accomplish a composite task or the possibility to script applications. As such we have become imprisoned into the interface as designed by the developer. Most applications are also designed to be used through a keyboard and a mouse even if the user has newer input devices. In this paper, we demonstrate how we can remove most of these limits and provide the possibility to script, adapt and automate GUI applications using a software bus in a Linux operating system. We provide implemented proof-of-concept cases in addition to conceptual scenarios showing the possibilities arising from the approach.

Categories and Subject Descriptors

H.5.2 [**Information interfaces and presentation**]: User Interfaces; I.3.6 [**Computer Graphics**]: Methodology and Techniques

Keywords

Operating system, interaction, architecture, post-wimp, factorization, automation

1. INTRODUCTION

The UNIX operating system has introduced the successful concept of building all the system experience by using small command-line tools which can be connected together through pipes. Unfortunately, this concept of "do one thing and do it well" is only available through the console and can not be applied to the GUI applications which have become more complex in each iteration. These applications are actually used as separated islands. They have some similar functionalities and with different and redundant implementations that forced special users as designers to use a huge list of tools in order to accomplish a bigger task. In this paper we are trying to solve the problem of GUI application complexity, inability of the reuse of specific functionalities, and inflexibility by introducing a new concept of communicating between the GUI applications, and describing scenarios that show the importance of such a mechanism. The scenarios we are exposing show numerous cases where the possibilities exceed the communication with a single application.

Our contribution mainly lies in the use of a software bus to ensure the applications connection and the ideation of new scenarios offered by this technique. Scenarios are a ground point to rethink how application could be architected in a common open source platform like Linux and evaluate the approach by enumerating and testing the feasibility of scenarios. We target creative tools and graphical design software used mainly by designers, novice or advanced. First, we start by presenting the concept of reducing the use of creative applications to their canvas and presenting the reasons which led to such a decision. Then we start presenting scenarios sorted in terms of complexity.

2. APPLICATIONS AS CANVAS

2.1 Definition of a Software Bus

A Software Bus is an inter-process communication system which allows different applications to connect to a shared channel and then communicate information or invoke distant method calls. In the case of Linux systems, D-Bus is used to facilitate the communication of three message types: method calls, signals and properties. It is possible to call a method in a remote application, subscribe to signals events emitted or read variable properties shared on the bus. As defined by the standard, each application has an address and a list of methods which can be introspected by any application reading from the bus. The choice of D-Bus is suggested by the number of tools, libraries and applications that already support it.

2.2 Previous and Related work

When Linux developers have finished the work on supporting multi-touch events inside Linux kernel, they faced the problem of modifying a big list of software libraries and layers to support the event routing between the kernel and the application. A faster solution developed was Ginn, which

.

is a Linux daemon that transforms multi-touch events into keyboard taps and mouse clicks. The transformation rules named "wishes", are loaded depending on the application currently in focus. The concept of transforming input events is similar to the one of the iCon project by Dragicevic [5]. A small piece of code enabled legacy applications to have a feeling that they react to the new events, without modifying the code of the legacy applications or soliciting the developers to do so. Ginn proposed a solution to adapt old applications to new ways of interaction, but it still needs to have the application in focus, and the old interface in which events are injected. From this case, we felt the importance of including pieces allowing applications to evolve in the future to support new means of interaction as referred by Chatty [4] than keyboard and mouse events. Ginn could do a better job if it were able to trigger, by a certain rule or wish, an internal function of the application. In this case, it would bypass the proposed GUI interface and only need the application canvas where the command response is shown as the case in Figure 1.

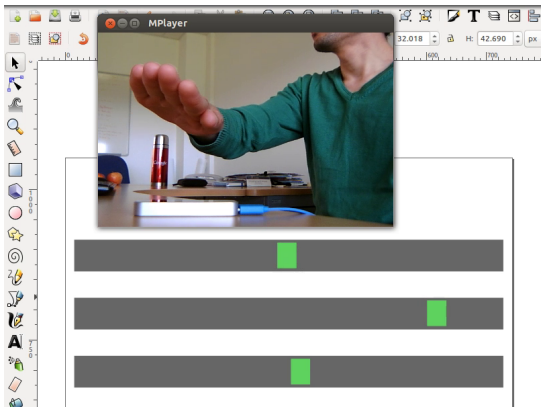

Figure 1: Connecting hand palm angle rotation events to control the movement of boxes inside inkscape, no modification to the target application code is needed

2.3 Reducing applications into their canvas

GUI applications have eliminated the need to memorize commands in order to accomplish tasks. Meanwhile, they have removed the possibility of scripting and intercommunication. Some attempts have tried to use software buses like TUIO [6] and IvyBus [3]. In the first case to transport input events which are more complex than the one standardized in the operating system, and in the second case to communicate between application to transport a generic type of information. Olivo et al. [8] have pointed out this problem while trying to handle multi-touch events, but used network protocols. That does not completely solve the problem, especially when many applications do not use the same platform standard. We also present how we can reduce the dictatorship of the applications interface on what the user is able to do. This is different of what has been done by Stuerzlinger et al. [9] by trying to innovate at the interface level in order to reach the same goal. Minimizing the role of most of the applications into a visualization canvas of what is being done by accessing its internal functionalities from a software bus. It can also simplify the application migration into a new post-wimp paradigm because we will only need to connect new input events into these internal methods and need the canvas only for feedback. Contrary to model-view-controller pattern where frontiers of a single application are known, we are addressing a platform with multiple applications that can be used on multiple devices which is not limited to original developers intended use. In most of the upcoming scenarios, we target the vector drawing application Inkscape and show what are the new possibilities arising from it.

3. SCENARIOS

3.1 Scripting GUI Applications

Graphical applications can only be used through their interface, and using the input devices supported. When we want to handle a complex manipulation we are stuck in repeating input actions, especially when developers have not provided neither a macro recorder nor an app-internal scripting API like VBA or Gimp PDB (Procedural DataBase). Some of the solutions to this problem is to use an external daemon for input record and replay, or more complex ones like MIT Sikuli [10] use computer vision to detect the position of graphical interface elements then generates clicks on them. Using computer vision for such operation means that we still need the graphical interface on the screen. This is more like a hack than a permanent solution, as the interface can change anytime between versions. In our case, by accessing the application methods exposed on a standardized software bus, we are able to script all needed actions using a simple python script executed from the outside of the application itself which is then transformed into a visualization canvas.

3.2 Interactive Programming

When we script application as we discussed, we also have the possibility to combine useful scripts, make them configurable and create a GUI for them. By taking the example of a drawing application like Inkscape, the application is divided into a canvas, where elements are drawn, and a default user interface. We can use floating windows in the same way of Magic Lenses by Bier et al. [2] to add a new functionality to the application. The floating windows will internally contain a script to handle an object inside Inkscape as in Figure 3. The window can use sliders to configure the values to draw a complex shape, and generate the DBus commands in order to stream them to the application. From simple application methods like "draw line" or "draw circle ", we can write external plug-ins able to draw any shape. They will communicate with Inkscape using the software bus, developed in any selected language and are solely limited by the imagination of this plug-in developer.

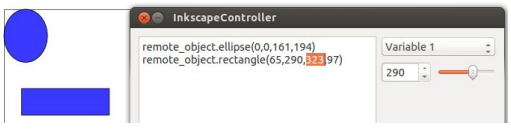

Figure 2: Implementation of an interactive geometric shapes drawing. The user can select a value and visualize the change instantly on Inkscape while moving the slider.

This new way of adding features to applications is very generic in terms of the programming language used or what is possible to, and can force the application to behave in newer ways. Figure 2 shows the use of C++ or python languages to add interactivity. For example we can make a drawing application to behave as a plotting tool by reading values from a file and transforming them into drawing commands to the app. We have even created an animated visualization by using the "move" method on a graphical element inside the plug-in internal loop which can export a frame each time then combined to create an video. Inkscape gained the animation ability just by plugging an external application that is based on its methods and compose them to create a new environment.

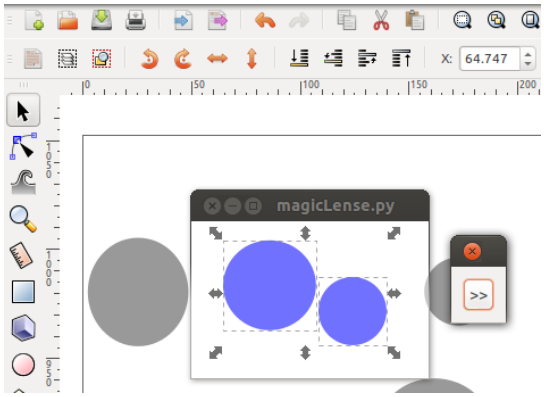

Figure 3: Implementation of a "magic lens" example. The completely separated application has a semi-transparent window and can modify the elements color in Inkscape

3.3 Factorization of applications development

In the previous example, we reached the level where we can make a normal application behave in new ways just by using external plug-ins that do not depend on the provided interface. This means that the important part of the application is changed. What are the limits if we push this concept into a new level by completely removing the default interface and providing new ones?

Any designer use many tools to create a mock-up or an animation. Tools he uses do not always come from a single company. Thus he needs to learn to use the interface of each of them. A beginner will find that these tools do not share the same icons, the same interaction design, even the same positioning of toolbar elements which is a problem of inconsistency in the actor platform. This problem can be solved when the developer of a platform can access internal functions through the installed software and provide by himself an interface which contain the same set of icons and based on a the same interaction design. And such an access can be done using a software bus between applications. We are proposing a resolution of a single platform inconsistency. But nowadays, applications now can be run on mobile phones, tablets, desktop computers and TVs. Even if the applications type can differ from one device to another, we still have a common space of functionalities used. In order to target all of these devices, a typical developer would create an application for each of them, with the proper interface and interaction, compile it and release a new version for each device. The problem here is that a lot of work and multiple skills from development to design are needed. When the person porting an application to another device is different, it becomes sometimes difficult to convince the main developer to create a new interface. This scenario applies the two levels of separation: Core functionalities including a visualization canvas, and a superposed interface. The core exports the internal functionalities into a software bus, and the interface connects to that bus, loads a selected interface and a "functionalities matching process" depending on the device. This concept is represented by Figure 4. The interfaces are created by main platform developers and the intellection will move from application level to platform level. The amount of work needed to adapt the application to a new device or modality is reduced to the creation of the interface and the matching. The core application role will be to show a canvas, and to export its internal methods into the bus. Here in addition to the proposed inconsistency resolution for single and multiple platform, we have factorized the application development into core functionalities which accelerate the development, and factorized also the interaction design which will be made by the designer for all the platforms. The link between the two layers is a component that matches the interaction to the functionality. Thinking about the platform and not the application itself, can push the thoughts into the solution of showing the canvas of the same application through many connected devices and being able to modify elements at the same time using the computer, the tablet and the phone, do part of the work on a device and finish it using another one. Events will be handled in each device and transformed into DBus method calls.

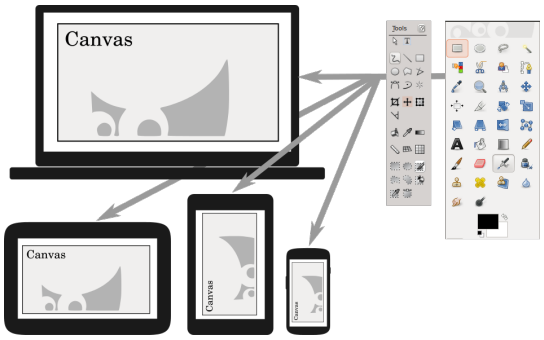

Figure 4: Concept of multi-device application development where the application will provide only a drawing canvas and export its functions through the bus

3.4 Applications Composer

In a platform where many applications get their internal methods exported and handled with the same way of thinking, we need a tool that is able to connect to any internal method of a chosen target application. Then run more complex tasks than what is provided by a single application. This tool capable for example to open a file inside an application as the "open file" is an exported method, apply operations on it and save it in a external file. Then, after completing with the first application, open these in

a new one and so on. This tool will make GUI application as powerful as UNIX command line tools with a lot more possibilities. Since it looks like Apple Quartz Composer, but using applications as building blocks, let's call it "Applications Composer". This concept enables applications scripting, and simplifies complex tasks for the user. It reduces the meaning of having many alternative tools to do the same task inside a platform. If we have many drawing applications that do almost the same thing, the composer interface eclipses these applications the way of independent units they used to be. A simple user who wants to accomplish a task will load a script, provide the input files and get the output, with no need to look at how this is done in details and what applications will be used in the chain. Some developers can even develop small applications with no interface and no significance when used alone, but that are here to fill the gap when used inside the composer. We are speaking about interaction with application using tools, without being limited to a specific interface this is a way to rethink the interaction in a platform as multi-device and multi-application, but single interaction design [1].

3.5 Interactive Documentation

The current way to create a documentation for a GUI application is either to write textual tutorials accompanied with pictures of the buttons where to click, screen-shots of the application windows, or to record a video of the author using the application directly. In both ways, the learning user should switch between the textual or video tutorial and the real application many times and to follow step by step what is shown. The switching is intensive and the user can easily lose focus and get lost as described by Laput et al. [7] Using the core model of a software bus, documentation can be built independently of the type of the current interface visible to the user, but on the core functions of the application instead. We have newer way to create documentation with DBus. For example, we can invoke an action, and we can also detect when a user does a specific step in a tutorial using signals. These can be used to show a part of the tutorial and go to the next step only when the user has done the previous required work. Using transparent windows, the user will no more switch between the application he is learning and the tutorial. The latter will be shown on top of the application step at-a-once that can even be written inside the learning document for a drawing application. Using this scenario, we have removed the switch between an application and its tutorial. The old forms of documentation can still be generated using a screen-shot taking or video recording script and will take into account the current interface according to the platform.

4. CONCLUSION AND LIMITS

In this paper we have presented many scenarios showing how the export of an application's internal methods can be beneficial for new platform design possibilities. Using the same concept we have also proposed solutions for problems like inconsistency in the interface and interactions of a platform. It also addresses the reduction of development time, multi-device deployment, unification of the documentation system. A side effect of our approach is the reducibility of creative creation GUI applications into their bare canvas. We have explained how this brings better interactions with computers for the developer who will write less code, as well as for the user who will get his task accomplished by writing small scripts. Developers do not want their tools eclipsed and hidden below a platform made, application composer. This also leads to the use of their applications in new, unintended way, which needs some standardization efforts like the MPRIS for video players.

5. REFERENCES

[1] M. Beaudouin-Lafon. Designing interaction, not interfaces. In *Proceedings of the Working Conference on Advanced Visual Interfaces*, AVI '04, pages 15–22, New York, NY, USA, 2004. ACM.

[2] E. A. Bier, M. C. Stone, K. Pier, W. Buxton, and T. D. DeRose. Toolglass and magic lenses: The see-through interface. In *Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '93, pages 73–80, New York, NY, USA, 1993. ACM.

[3] M. Buisson, A. Bustico, S. Chatty, F.-R. Colin, Y. Jestin, S. Maury, C. Mertz, and P. Truillet. Ivy: Un bus logiciel au service du développement de prototypes de systèmes interactifs. In *Proceedings of the 14th French-speaking Conference on Human-computer Interaction (Conférence Francophone Sur L'Interaction Homme-Machine)*, IHM '02, pages 223–226, New York, NY, USA, 2002. ACM.

[4] S. Chatty, A. Lemort, and S. Vales. Multiple input support in a model-based interaction framework. In *Horizontal Interactive Human-Computer Systems, 2007. TABLETOP '07. Second Annual IEEE International Workshop on*, pages 179–186, Oct 2007.

[5] P. Dragicevic and J.-D. Fekete. Support for input adaptability in the icon toolkit. In *Proceedings of the 6th International Conference on Multimodal Interfaces*, ICMI '04, pages 212–219, New York, NY, USA, 2004. ACM.

[6] M. Kaltenbrunner, T. Bovermann, R. Bencina, and E. Costanza. Tuio: A protocol for table-top tangible user interfaces. In *Proc. of the The 6th IntâĂŹl Workshop on Gesture in Human-Computer Interaction and Simulation*, 2005.

[7] G. Laput, E. Adar, M. Dontcheva, and W. Li. Tutorial-based interfaces for cloud-enabled applications. In *Proceedings of the 25th Annual ACM Symposium on User Interface Software and Technology*, UIST '12, pages 113–122, New York, NY, USA, 2012. ACM.

[8] P. Olivo, D. Marchal, and N. Roussel. Software requirements for a (more) manageable multi-touch ecosystem. In *EICS 2011 Workshop on Engineering Patterns for Multi-Touch Interfaces*, 2011.

[9] W. Stuerzlinger, O. Chapuis, D. Phillips, and N. Roussel. User interface façades: Towards fully adaptable user interfaces. In *Proceedings of the 19th Annual ACM Symposium on User Interface Software and Technology*, UIST '06, pages 309–318, New York, NY, USA, 2006. ACM.

[10] T. Yeh, T.-H. Chang, and R. C. Miller. Sikuli: Using gui screenshots for search and automation. In *Proceedings of the 22Nd Annual ACM Symposium on User Interface Software and Technology*, UIST '09, pages 183–192, New York, NY, USA, 2009. ACM.