Add some Olives to your coffee: A Java-based GUI for the Octopus system

Aram Sadogidis, aram@privatdemail.net
Spyros Lalis, lalis@inf.uth.gr

University of Thessaly
Volos, Greece

October, 2012

ABSTRACT

In this paper we describe our efforts to enable Java supported terminals to interact with the Octopus pervasive environment. In order to achieve that goal, we developed a Java-based implementation of the Octopus GUI front-end.

1 Introduction

Pretty soon many people will own a large number of smart devices with Internet capability through the wire or over the air. While having many gadgets can be great fun, the user will also be faced with the burden of managing a highly decentralized, uncoordinated, heterogeneous and dynamic device ecosystem. Currently, the problem is typically "solved" by letting the user act as a mediator between these devices. This approach, apart from not being much fun, cannot possibly scale to a large number of devices. As another, relatively recent option, data and applications can be placed in the cloud. However, the flexible exploitation of the hardware and software resources of other devices remains a challenge.

The Octopus system [1, 2] tackles this problem by applying the principle of centralization (as the cloud paradigm) in conjunction with an open, simple and flexible resource sharing architecture. All applications run in a single computer, called ***the computer***, while every other smart device connects to the computer over the network to provide special resources or act as interactive terminals for these applications.

Like Octopus itself, the standard GUI terminal support for the Octopus is implemented on top of Inferno [3]. Even though Inferno has been ported on many platforms (e.g., Linux, Windows, MacOSx, Plan9, Solaris and BSD), in practice only a few people will go through the installation process merely so that they can run an Octopus terminal. Moreover, for all practical purposes, one cannot expect that Inferno will or should be ported on all platforms. However, the Java runtime environment is being ported, aggressively, on all kinds of platforms and enjoys strong support from a large part of the industrial world. In particular, a significant share of the booming smartphone market goes to Android, which comes with native Java support.

With this rationale, we think it is a good idea to extend the arms of the Octopus so that it can reach out for Java terminals. Aiming for that goal, we developed a Java-based implementation of the Octopus GUI front-end, named JOlive (after Olive), through which one can interact with all the applications running on an Octopus computer (which can run anywhere in the net/cloud). In addition, we ported JOlive on Android thereby enabling a smartphone to act as yet another terminal device for the Octopus, while keeping the native look & feel. Besides dealing with issues specific to the Android environment, the smartphone version addresses the problem of limited screen real-estate, by allowing the user to "pull" the UI of just a few or perhaps even a single application in order to enjoy a more focused and efficient interaction.

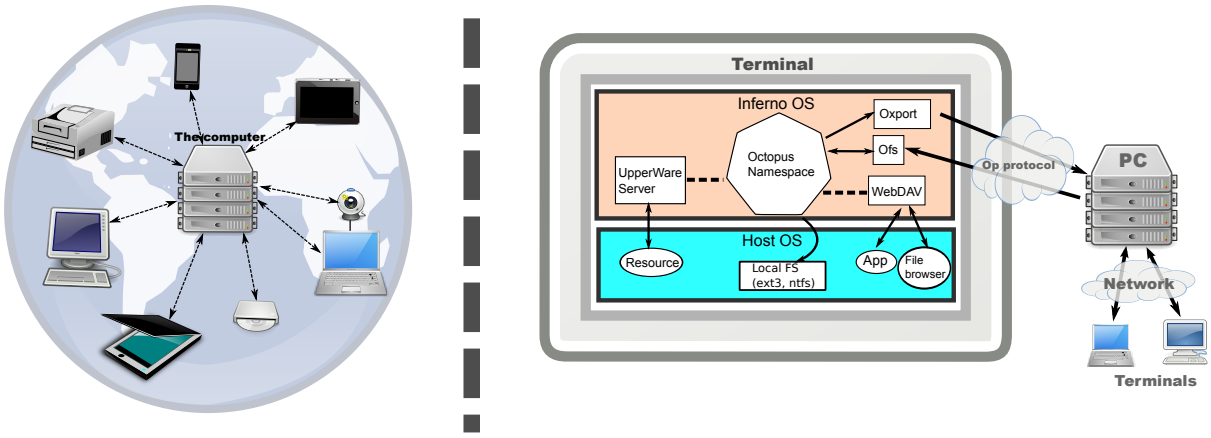

Figure 1: Various terminal resources connect to the central PC and export resources.

In the following we give a brief overview of Octopus and the Octopus UI system. Then, we proceed to describe our Java-based implementation and the respective Android port. We close the paper with some reflections on future work.

2 The Octopus in brief

The Octopus system [1, 2] aims to provide a single and homogeneous (for the application programmer) yet heterogeneous and ubiquitous (for the user) multi-device computing environment. Rather than pretending that all devices are equals, and trying to support a peer-to-peer interaction between them, Octopus distinguishes a single node, called ***the computer***, as the center of the personal computing universe, where everything else is connected to. Figure 1 shows an example. The connections between all other devices and the computer are implemented using the Styx [4] and Op [5, 6] protocols (the latter being designed for long-latency links), which support the abstraction and access of resources in the form of synthetic file systems.

All applications run on the computer. Hence the user is relieved from the frenzy of downloading, installing and managing different applications (including their state and data) on different devices. Moreover, the user can attach, at any point in time, various devices to the computer, in which case their resources automatically become visible to the applications running there. While switching-off devices or bad connectivity will lead to the loss of these resources, such mishaps do not lead to nasty failures or a corruption of crucial application runtime state.

Octopus is built on top of Inferno [3] which in turn runs directly on bare metal or hosted on popular operating systems. Indeed, this large host base is exploited to implement a so-called UpperWare approach [7] for connecting legacy devices to the Octopus computer. More concretely, an Inferno-based software layer, placed on top of the native OS, abstracts and exports various hardware and software resources of the device in the form of a synthetic file system that speaks Styx or Op; of course, local access of these resources occurs though the host OS. This concept is illustrated in Figure 1. It is worthwhile noting that legacy applications can also be wrapped as resources (provided they can be accessed through the host OS interface), making it possible for them to be exploited from within Octopus applications and, conversely, applications (or the user) can exploit Octopus resources through a file system interface.

One example of an UpperWare device exploiting a host application resource is the browserfs [8, 9]. Its purpose is to control the web browser of a terminal in order to replicate its state (bookmarks, history, etc) to other terminals. The UpperWare driver, with the support of host command scripts, stores in regular disk files the state of the browser. These state files are accessible through the Octopus namespace by other terminals, who re-export them to their

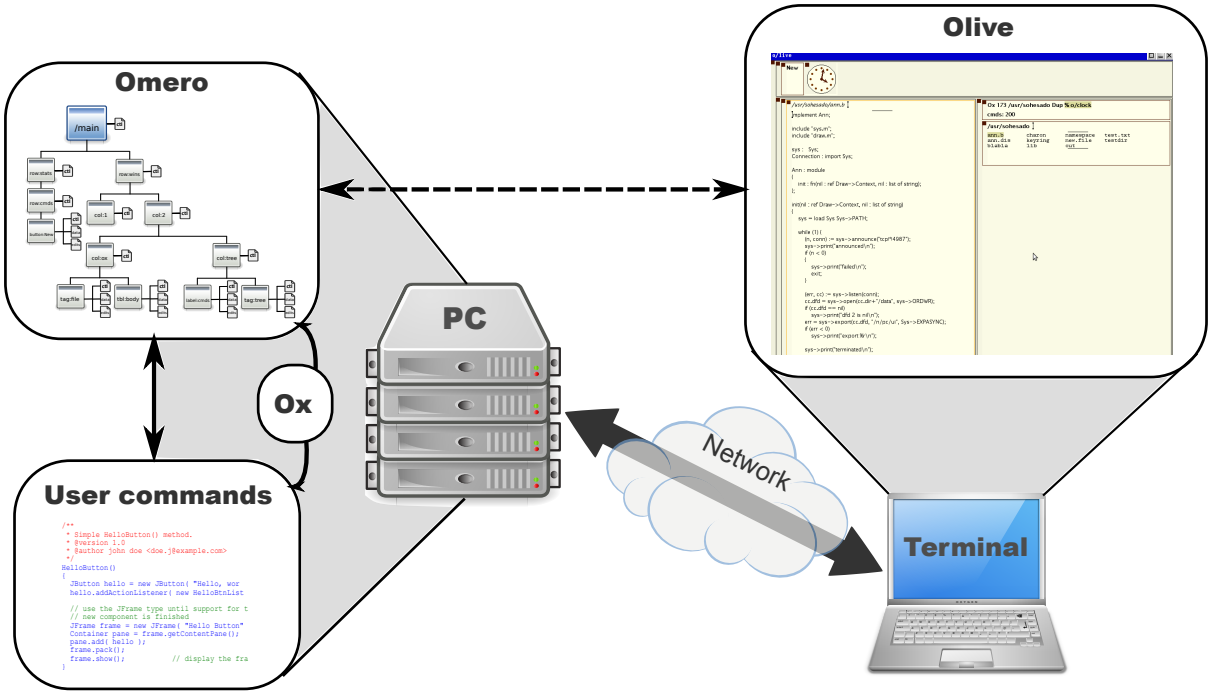

Figure 2: Example of a tree served by Omero and used by Olive on an Octopus terminal.

host systems over WebDAV.

Octopus employs the file-system abstraction for yet another purpose, namely to implement its window system, called Omero [10]. Omero serves a file tree with separate directories representing different virtual screens and a hierarchy of graphical components, known as ***panels***, each being represented in turn by a directory with a control file, and other optional files depending on the type of the panel. A typical Omero file hierarchy is shown in the left part of Figure 2.

Notably, the architecture of Omero enables a number of interesting functionalities, such as: (i) ***migrating*** any part of an application's UI to any terminal simply by moving the corresponding filesystem hierarchy branch to the directory for the screen assigned to this device; (ii) ***replicating*** parts of the UI on different terminals by copying panel directories to multiple screens; (iii) ***reusing*** powerful tools, e.g., ***tar*** to save and restore UIs, or ***find*** to locate windows with a certain title; and last but not least (iv) offering a ***familiar and easy-to-use API*** for writing GUI applications without bothering the programmer with the actual physical placement of the UI on one or more screens.

Omero does not draw and does not interact with the user. This functionality is delegated to the viewer component of the window system, called Olive [10] which runs on terminal devices. Olive is the only program that knows how to draw, how to interact with the mouse and the keyboard, and how to graphically manifest different types of panels. It initializes the graphical window based on the virtual screen to which it is attached to. From that point onwards, it maps the local keyboard and mouse events to the corresponding file operations on Omero's panels, and receives notifications about any UI updates through the Omero event channel file.

Finally, Octopus comes with its own shell, called Ox [11], which can be used to browse the file system, edit, and execute commands. The command language of Ox consists of some built-in commands, Sam commands, host OS commands and Inferno commands. The user can type and execute these commands explicitly, or through the tag and panel operations of the window system viewer.

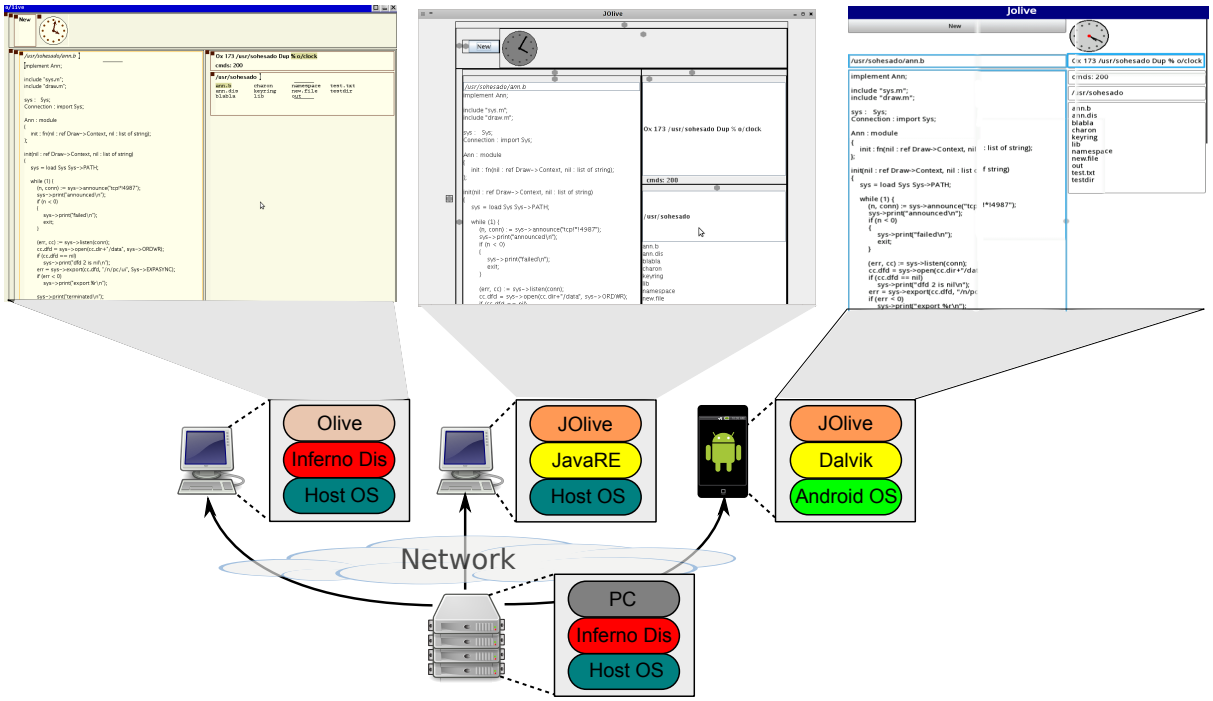

Figure 3: The three terminal variants with their respective UI screenshots.

3 The Java Octopus Terminal

Arguably, the Octopus system's effectiveness increases proportionally to the number of resources integrated to the global namespace. In order to broaden the spectrum of potential resources, we have implemented a Java-based GUI for the Octopus system. By doing so, we enabled Java-supported devices to export display resources (i.e. screen space) to the Octopus environment. The current implementation can be extended in order to expose other resources too (Section 4).

There are quite a few advantages of a Java-based approach compared to the original Inferno--based. A Java developer has the luxury to choose from many GUI toolkit options (AWT, SWT, Swing, Apache Pivot, JavaFX, Qt Jambi) whereas an Inferno developer's options are rather limited. One other technical advantage is the easier installation of Java virtual machine and its applications. Also there are available additional JavaVM-based languages to choose from, like Scala, Jython, Groovy, etc.

It should be noted that by including the Java platform as an alternative underlying software technology, we aim to expand the Octopus development opportunities, not to replace the Inferno-based terminal. Having said that, the number of mobile platforms supporting Java is an "audience" that should not be easily neglected.

JOlive is an Omero tree ***viewer***, implemented in Java. The first version we've developed is based on Java Standard Edition version 6 (JavaRE-1.6 virtual machine) and the Swing graphical toolkit. The second version is focused towards the Android 2.2 platform and therefore it is based on the Dalvik virtual machine and the Android graphical toolkit. Both versions use JStyx[1] to communicate with Omero. The Figure 3 provides an overview of the two implementations compared to ***Olive***.

[1] JStyx's port to Android required some tweaking (Section3.2.3).

OPanel	An object of this class corresponds to a directory that represents a panel. It contains the functionality to invoke ***ctl*** commands, retrieve ***data*** and panel related ***attributes.***
OOlive	This class represents the ***olive*** file served by Omero. It runs on a dedicated thread and it is responsible to receive update messages, demultiplex them and pass them to the corresponding JOPanel object.
Merop MeropCtl MeropUpdate	These three classes encapsulate the information related to the event messages generated by Omero. MeropCtl and MeropUpdate are derived from Merop, effectively unpacking/packing the messages to a ***ctl*** or ***update*** type.
OAttrs OUtils	OAttrs encapsulates the attributes that a panel may have and OUtils contains debugging functions.

Table 1: This table presents the omero package class description for the desktop JOlive.

3.1 JOlive for the desktop

JOlive maintains a bidirectional communication with Omero that implements the action-effect feedback loop. There is the ***user action notification*** which occurs whenever the user interacts with the GUI and the ***UI update notification*** whenever a change occurs to the Omero synthetic file system. The first data flow updates the Omero tree based on the user's actions, and the second updates the terminal's GUI based on events generated by the window system.

The graphical components presented by JOlive, have the corresponding ***action event*** listeners which invoke the corresponding remote commands. So, in an event-oriented fashion the Omero tree is updated as soon as an event is triggered by an action (e.g. a button click).

The ***update notification*** data flow path is more complex. When the filesystem is updated, either by an application or a viewer, Omero generates event messages which mirror the changes, encoded according to the merop protocol[2]. A dedicated component is required to receive the update messages and demultiplex them. The received data are packed ***merop*** messages that are unpacked into ***Merop*** java objects.

The implementation is composed of two packages. The ***omero*** package which implements the synthetic file system communication part and the ***ui*** package that implements the visual part of JOlive. The first is described in table 1 and the second in table 2 and the relationship between the classes in Figure 4.

3.2 JOlive for Android

In order to port JOlive to the Android platform we had to port the JStyx library first. After some hacking, we ported it successfully and consecutively we reused, almost intact, the ***omero*** package which is the JStyx dependent part of the implementation. On the other hand, we had to reimplement the ***ui*** package from scratch since the Swing API is not available on the Android platform. A side benefit of the reimplementation was that the resulting GUI obtained the native Android look & feel.

The Android version follows the same design with the Desktop version. The **additional** classes that have been implemented, are listed table 3.

[2] Omero defines a data packing/unpacking private protocol in order to transmit messages to Olive.

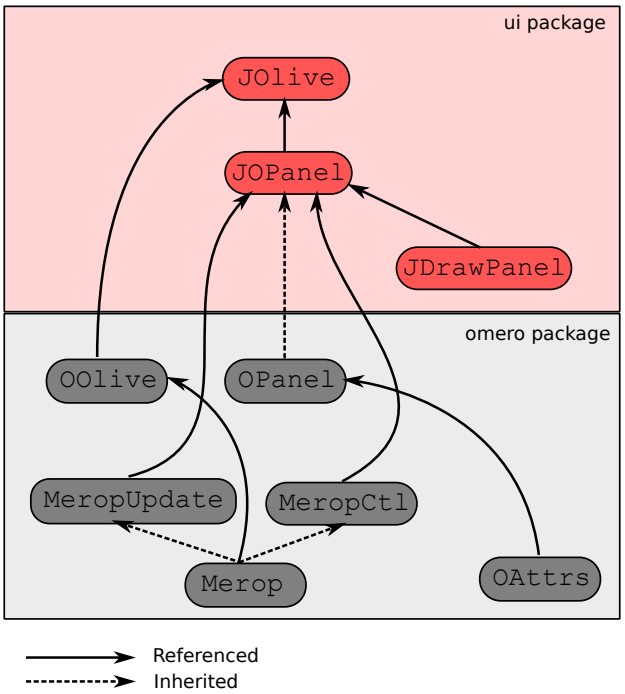

Figure 4: The class dependency of the implementation.

JOPanel	This class is a direct descendant of OPanel and has a reference to the graphical component corresponding to the Panel. Every object of this class is registered to the OOlive's hashtable in order to be notified if a related update message arrives. In essence this class combines the JStyx related stuff with graphical part of the implementation.
JDrawPanel	This is a customized JPanel component, used to represent the Omero's draw panel.
JOlive	The main class that initializes the window and initiates a connection with the Octopus PC.

Table 2: This table presents the ui package class description for the desktop JOlive.

OOliveEventHandler	This is the only additional class in the omero package used to tackle a technical difficulty presented by the Android runtime (Section 3.2.3).
ActionSwipeListener TextLongClickListener	These two classes implement the command invocation swipe widget (Section 3.2.1)
PullAppListener	This class implements the Pull App functionality (Section 3.2.2)
ConfigureLog4j	This is a hack employed while porting JStyx to Android (Section 3.2.3)

Table 3: This table presents the ui package class description for the desktop JOlive.

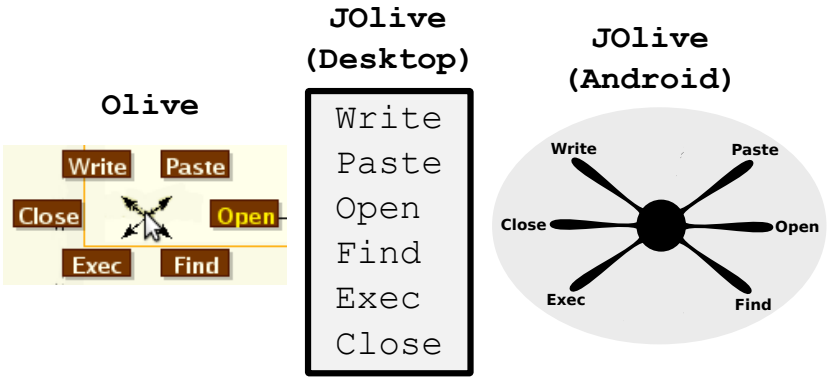

Figure 5: The three variants of menu graphical components.

3.2.1 Command invocation swipe widget

The user may click on a margin or tag and invoke commands over the Octopus panels. Although for the desktop version we've followed the traditional right-click drop down menu, for Android this approach was dismissed as it was rather impractical for touchscreen devices. Instead we adopted the exact same solution employed in Olive, namely the popup menu that shows different options in a circle around the point (refer to Figure 5). To select one you must move the pointer quickly in the direction of the option.

For the Android implementation, in order to invoke a command the user swipes his finger from the center to the direction of the desired command. In essence, the mouse gesture used in Olive, is replaced with a swipe action on the touchscreen. From the usability efficiency point of view, improvements like this make the difference because the menu invocation is the most frequent action that an Octopus terminal user employs. The three menu variants are depicted at the Figure 5.

On the technical side, this widget is implemented by a PopupWindow overlaid with a customized ImageView that tracks finger swipe actions. The widget is generated when a LongClick occurs and destroyed when the chosen command is issued. The swipe gestures can be modified easily by editing the `ActionSwipeListener.java` file in case tweaking is required to accomodate a certain touchscreen's configuration.

3.2.2 Pull function

While the user may have numerous, many tens of open apps, he may wish to use the terminal to interact with just a few or perhaps even just one app at a time, this is especially meaningful when using small screens. From this observation emerges the need of a functionality to easily isolate a small number of application UIs. We've implemented the ***Pull*** option for the Android version which helps the user to select a subset of the system's GUI.

Consider the following motivating scenarios. I want to use my mobile phone to browse and add an appointment in my calendar that is displayed on the main screen, along with my other apps. Instead of scrolling through the numerous running application panels, I ***pull*** the calendar related panels to the mobile phone's screen. As another example, imagine the user makes a presentation with a projector and desires to control the slides remotely. He can pull

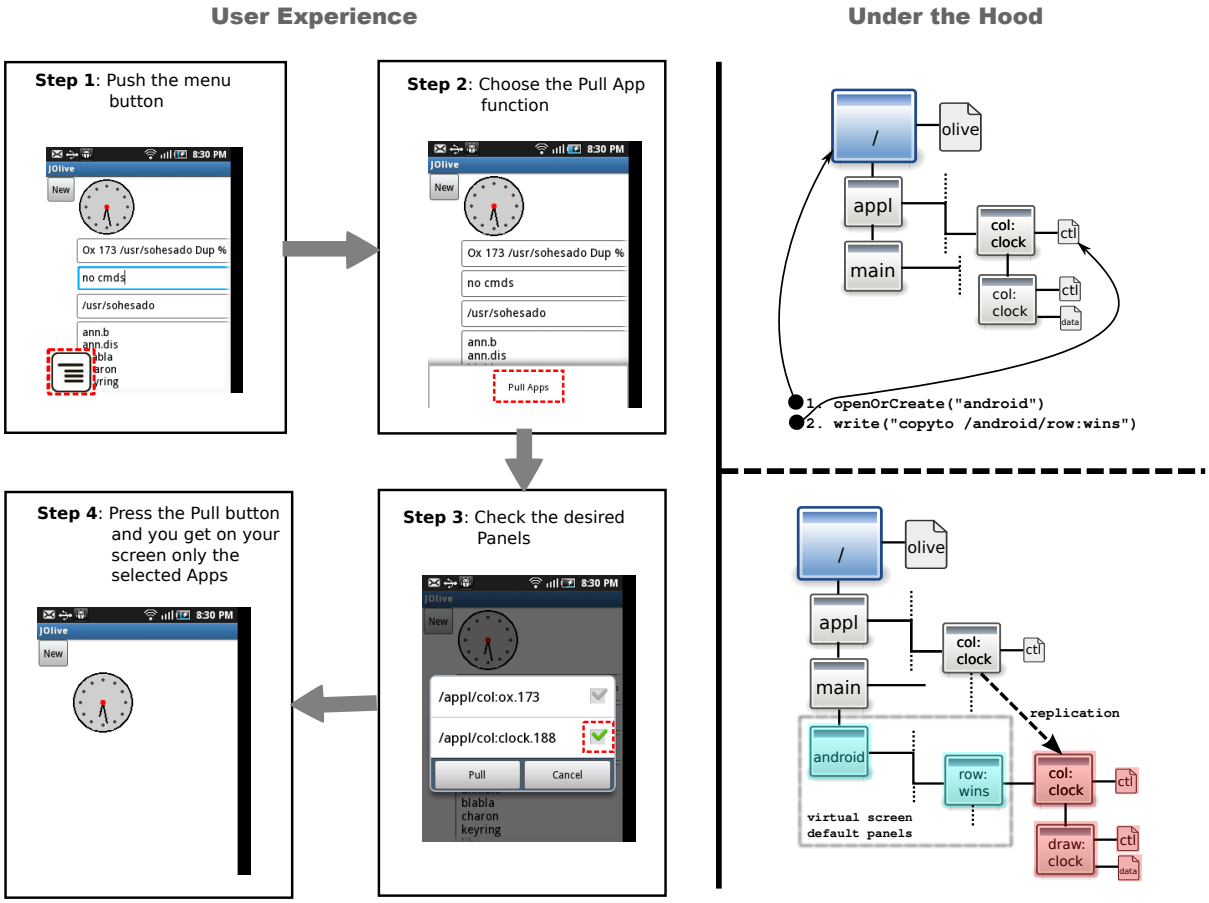

Figure 6: This Figure illustrates the Pull App functionality, at left the user's perspective, at right the underlying process.

the ***next/previous*** buttons to his Android device and use it as remote control. The same thing applies to the case of a multimedia player application or anything that has some sort of graphical control interface and makes sense the remote interaction with that app.

The approach that we adopted in order to implement the pull functionality is based on a popup dialog in order to assist the user to easily select panels. The procedure can be break down to the following steps. (i) Push the menu button. (ii) Select the Pull function. Pops up a checkbox list with the apps running at the PC. (iii) Check the desired apps. (iv) Touch the Pull button and as a result you get on your screen only the panels related to the selected applications. In Figure 6 is illustrated the process of "pulling" the clock application.

Under the Hood, JOlive initializes a new virtual screen by creating a directory at the Omero's root and consecutively issues ***copyto*** [11] commands to the selected panels. Omero copies the file descriptors to the supplied path and generates the corresponding update events. Once JOlive receives that event messages, it presents the newly initialized virtual screen. After the pull is invoked successfully, it will receive updates only for the panels present to that virtual screen, which makes it more efficient.

This effect can be achieved by issuing the appropriate Ox commands. However the extra 30 lines of code or so required to implement this function via a proper dedicated UI element are justifiable in order to avoid the tedious task of typing via touchscreens. With this option, the user with a couple of clicks achieves the result of textual commands that had to be passed manually to Ox.

3.2.3 Trivia

We faced some technical challenges while porting the implementation to the Android environment. While porting JStyx to Android we faced a technical problem related to log4j message logging package because it was incompatible with the Android platform. We had to switch to the compatible alternative that required the android-logging-log4j and slf4j-android along with ConfigureLog4j.java code that configures said packages.

One other unexpected problem caused by the Android runtime environment was the limitation that only the thread that creates a ***View*** object, is allowed to update it. In our original design, the main thread initialized the graphical window (i.e. created the graphical widgets) and a secondary thread received Omero event messages and updated accordingly the respective widgets. This design caused runtime errors so we had to approach the problem in a different angle. We used a ***handler*** object created in the main thread that was asynchronously receiving update messages from the OOlive thread (Table 1) and updating the GUI accordingly.

4 Future perspective

JOlive is an Omero ***viewer*** which is the most important component of a terminal, since it provides the means to interact with an Octopus system. The following subsections describe briefly some interesting resources and other terminal related extensions that can be implemented in order to increase the effectiveness of the Java terminals.

4.1 GPS resources

Modern mobile phones have GPS capabilities. It seems promising, from a pervasive computing environment point of view, to expose the GPS tracking service to the system. The Android terminal could export its GPS coordinates through a passive UpperWare resource driver to the global Octopus namespace, with the intention to enhance the smart space characteristics of the system. Assuming the user carries his phone with him, the system could then automatically infer the location of other mobile computing devices, such as laptops, cameras or other wearable sensors, which in turn could be exploited for context-aware computing purposes.

4.2 JOp

A Java implementation of the Octopus protocol seams interesting. Op is preferred instead of Styx when high latency communication links are encountered. Certainly, it would be interesting to create a Java implementation of Op, at least the client part, which is a useful tool on its own. Indeed, especially the Android version Java terminal could greatly benefit from an Op implementation, given that it will typically communicate with the Octopus computer over WiFi hot-spots or the cellular network (e.g. 3G data transfer network).

4.3 Remote control by voice

The Octopus experiments with the notion of integrating voice support to the system. Smartphones have decent voice recording capabilities. A possible extension of the Android terminal would be to implement the functionality to take advantage of that capability. If we are able to export the voice recording resource of our phone to the Octopus system, assuming there will be a decent audio command server implementation, then the Android terminal would offer many practical applications. In a "smart" space environment, but also when the user is on the move, it can be very practical to issue commands to, but also receive messages from the system, verbally.

4.4 Authentication device

We can exploit the habit of keeping the cell phone within our reach and create an authentication mechanism resource for the Octopus system. We can device a simple and secure mechanism for generating uniquely identifiable tokens that are hard to replicate without possessing a suitable hardware device. One possible way to do this is to combine a user defined touchscreen gesture, e.g., the user's signature, with the hardware ID of the device and generate a digest value that authenticates the user's credentials. The digest value does not give away neither the gesture nor the hardware ID and can be stored to the PC. Even if the secret gesture is "leaked", the token can't be generated without the hardware ID of the user's phone, and vice versa. This resource can be wrapped with a passive UpperWare driver and exploited by other system resources that require a more secure authentication mechanism.

References

[1] Francisco J. Ballesteros, Spyros Lalis, and Enrique Soriano. Building the Octopus. GSyC Tech. Rep, 2006-06.

[2] Francisco J. Ballesteros, Pedro de las Heras, Enrique Soriano, and Spyros Lalis. The Octopus: Towards building distributed smart spaces by centralizing everything. UCAMI, 2007.

[3] Sean Dorward, Rob Pike, David Leo Presotto, Dennis M. Ritchie, Howard Trickey, and Phil Winterbottom. The inferno operating system. ***Bell Labs Technical Journal***, pages 5--18, 1997.

[4] Rob Pike and Dennis M. Ritchie. The styx architecture for distributed systems. ***Bell Labs Technical Journal***, 4(2):146--152, April-June 1999.

[5] Francisco J. Ballesteros, Gorka Guardiola, Enrique Soriano, and Spyros Lalis. Op: Styx batching for high latency links. IWP9, 2007.

[6] Francisco J. Ballesteros, Enrique Soriano, Spyros Lalis, and Gorka Guardiola. Improving the performance of styx based services over high latency links. Rosac, Laboratorio de Sistemas, 2 2011.

[7] Francisco Ballesteros, Gorka Guardiola, and Enrique Soriano. Octopus: An upperware based system for building personal pervasive environments. ***Journal of Systems and Software***, 85(7):1637--1649, July 2012.

[8] IEEE Middleware Support for Pervasive Computing Workshop (PerWare). ***Upperware: Bringing Resources Back to the System***, 2010. in proceedings of the PerCom 2010 Workshops.

[9] Gorka Guardiola, Francisco J. Ballesteros, and Enrique Soriano. Upperware: Pushing the applications back into the system. IWP9, 2008.

[10] Francisco J Ballesteros, Enrique Soriano, and Gorka Guardiola. Towards persistent, distributed, and replicated user interfaces in the octopus. IWP9, 2007.

[11] Laboratorio de Sistemas. ***Octopus 2nd. edition User's Manual***, 1 2008. RoSAC.