Java implementation of the graphical user interface of the Octopus distributed operating system

Submitted by
Aram Sadogidis

Advisor
Prof. Spyros Lalis

University of Thessaly
Volos, Greece

October 2012

Acknowledgements

I am sincerely grateful for all the people that supported me during my University studies. Special thanks to professor Spyros Lalis, my mentor, who had decisive influence in shaping my character as an engineer. Also many thanks to my family and friends, whose support all these years, encouraged me to keep moving forward.

Contents

List of Figures

Chapter 1

Introduction

With the expansion of the Internet and the widespread usage of a variety of computing devices that are constantly evolving, emerges the need to establish a pervasive computing environment to satisfy the ever evolving modern user's demands. That need arises from the fact that the users face decentralized, uncoordinated, heterogeneous, and highly dynamic, practically uncontrolled environments. Octopus [1,2] is a system that aims to provide a single, homogeneous, ubiquitous computing environment by centralizing everything to a single personal computer. Other devices and services connect over the network, as terminal resource providers, to the central system. The aggregation of distributed resources in a unified, homogeneous environment, creates the illusion of a ubiquitous *virtual* computer which a major step into making pervasive computing a reality..

The resources that can be added to the Octopus pervasive environment have to conform to a number of restrictions which are imposed mainly by the software technologies upon which the system is built. The goal of this project is to expand the set of candidate system software technologies, by bringing into play the Java platform. To achieve that goal, we have to implement an Octopus terminal in Java, in order to be able to expose Java supported devices into the Octopus "ecosystem".

We developed a prototype Java implementation of the front-end of Octopus's graphical user interface, named JOlive. This is the biggest component of a full-fledged Octopus terminal, since it provides full control over the applications running on the Octopus environment. The functionality that it lacks, is to expose other resources than the display, to the system. We succeed to port JOlive to the Android platform which enables Android devices to display the user interface of an Octopus system. Additionally, for the Android version, we developed a functionality to manage the UI for mobile devices by flexibly *pulling* a selected subset of the complete GUI.

The outline of the document is:

2. **System software technologies** A brief description of the underlying technologies supporting the Octopus system. This knowledge is a prerequisite in order to understand the concepts presented later, because these are the building blocks of the system. These technologies both impose restrictions and offer opportunities for the development of the system.

3. **Octopus OS** This chapter describes the Octopus OS, covering a number of aspects like the UpperWare architecture and the windowing system. In order to appreciate the purpose of this project, one has to first understand how the Octopus system works.

4. **Java Octopus terminal** The JOlive implementation is described which encapsulates the software development effort made during this project. Also it contains some of the technical challenges faced during the development and the experience gained from that endeavour.

5. **Future Perspective** Few interesting opportunities arise from a Java terminal implementation. A couple of promising extensions are described, planned for implementation in the future.

Chapter 2

System software technologies

Octopus system is built out of UpperWare [3, 4], a design paradigm that wraps computing resources with a file system interface and aggregate's them to a single file system tree. This approach inevitably imposes the requirement to use a synthetic filesystem [5] protocol. If we combine this fact with the qualities that the system should satisfy, namely to be able to run on a variety of hardware and operating system configurations, Inferno OS becomes a strong candidate as a supporting platform. Indeed, Octopus is built on Inferno since it offers a built in synthetic filesystem protocol and it can run on a big number of platforms, both hosted on popular OSes and stand-alone on bare metal.

2.1 Inferno OS

Inferno OS [6, 7] is a distributed operating system implemented at Bell Labs [1] based on the experience gained with Plan 9 from Bell Labs. It can operate stand-alone (running directly on a hardware) or in a hosted environment. It defines a virtual machine, called Dis, which is implemented for all modern operating systems [2]. It has three design principles.

1. All resources are named and accessed like files in hierarchical file systems.

2. Disjoint resource hierarchies can be joined together into a single, private hierarchical namespace.

3. A single communication protocol, Styx, is used to access all resources, whether local or remote.

[1] Now it is developed and maintained by Vita Nuova Holdings as free software.
[2] linux, windows, macosx, plan9, solaris, BSD.

Inferno is intended to be used in a variety of network environments, for example those supporting advanced telephones, hand-held devices, TV set boxes attached to cable or satellite systems, and inexpensive Internet computers, but also in conjunction with traditional computing systems [8]. The qualities that enable this kind of versatility are:

Portability across processors: It currently runs on Intel, Sparc, MIPS, ARM, HP-PA, and PowerPC architectures and is readily portable to others.

Portability across environments: It runs as a stand-alone operating system on small terminals, and also as a user application under Windows NT, Windows 95, Unix (Irix, Solaris, FreeBSD, Linux, AIX, HP/UX) and Plan 9. In all of these environments, Inferno applications see an identical interface.

Distributed design: The identical environment is established at the user's terminal and at the server, and each may import the resources (for example, the attached I/O devices or networks) of the other. Aided by the communications facilities of the run-time system, applications may be split easily (and even dynamically) between client and server.

Minimal hardware requirements: It runs useful applications stand-alone on machines with as little as 1 MB of memory, and does not require memory-mapping hardware.

Portable applications: Inferno applications are written in the type-safe language Limbo [9], whose binary representation is identical over all platforms.

Dynamic adaptability: Applications may, depending on the hardware or other resources available, load different program modules to perform a specific function. For example, a video player application might use any of several different decoder modules.

Inferno applications can be written in C or in a new programming language introduced with the system, Limbo [6, 9]. The later is a type-safe language that is compiled to portable byte code and it is used for writing distributed systems. Octopus itself is developed in Limbo. It is syntactically similar to C, it has several features that make it simpler, safer and yet more powerful and better suited to the development of concurrent, distributed systems. The Limbo compiler generates architecture independent object code which is then interpreted by the Inferno Virtual Machine, named Dis, or compiled just before runtime to improve performance. This ensures that Limbo applications are completely portable across all Inferno platforms.

The Octopus system benefits from the capabilities of the Inferno OS in many ways. For one it satisfies the major requirement of a synthetic filesystem support. Also, every possible device that can host Inferno, is a prospective resource for Octopus. As it will explained later on, once you are able to run an Octopus terminal on a device, it is easy the expose underlying resources to the rest of the system.

2.2 JavaSE and Android framework

Java Standard Edition [10], is a programming platform which enables the deployment of portable applications for general usage purposes. It consists of a virtual machine (JavaRE) and a set of libraries. All popular operating systems have a JRE implementation which makes Java programs cross-platform. Even the web browsers are able to host Java programs, called applets. There are many Java-enabled platforms. This is a brief list of systems that have a JRE implementation.

- Solaris-OpenIndiana
- All Windows variants
- GNU \Linux distributions
- FreeBSD/NetBSD/OpenBSD
- Mac Os X
- Popular Web Browsers (Firefox, Chrome, Safari)

As a result, applications developed in the Java language, can execute on a vast number of computing platforms.

Along with the Java virtual machine, a collection of libraries is supplied too. The Standard Edition, which is used in this project, offers the following packages.

java.lang Contains fundamental classes and interfaces closely tied to the language and runtime system. This includes the root classes that form the class hierarchy, types tied to the language definition, basic exceptions, math functions, threading, security functions, as well as some information on the underlying native system.

java.io Contains classes that support input and output.

java.math The java.math package supports multiprecision arithmetic (including modular arithmetic operations) and provides multiprecision prime number generators used for cryptographic key generation.

java.net The java.net package provides special IO routines for networks, allowing HTTP requests, as well as other common transactions.

java.text The java.text package implements parsing routines for strings and supports various human–readable languages and locale-specific parsing.

java.util Data structures that aggregate objects are the focus of this package.

java.applet This package allows applications to be downloaded over a network and run within a guarded sandbox.

java.beans A package with various classes for developing and manipulating beans, reusable components defined by the JavaBeans architecture [3]

java.awt and javax.swing These two, provide access to a basic set of GUI widgets and a collection of routines to implement GUI interfaces.

java.rmi The java.rmi package provides Java remote method invocation to support remote procedure calls between two java applications running in different JVMs.

java.security Support for security, including the message digest algorithm.

java.sql For database implementations.

This is the programming environment that is supplied to a JavaSE developer, which directly affects the flexibility offered to the Java Octopus terminal developer.

In many ways, Inferno OS aims to solve the same problems as Java. In a more abstract sense, both define a virtual machine upon which portable applications can be deployed. But Inferno is a full-blown operating system with its own protocol stack and graphical user interface, whereas Java is just the virtual machine along with an API. To make the matter more complicated, there is an implementation of Java on the Inferno operating system which enables development and execution of Java applications within Inferno [11]. For the intents and purposes of the project described in this document, the portable nature of Java is what matters. If we combine it with a synthetic filesystem implementation we have a suitable platform for the development of an Octopus terminal.

[3] http://en.wikipedia.org/wiki/JavaBeans

One other benefit of a Java terminal implementation is that we gain the opportunity to take advantage of the popular Android OS[4] since the programming language of choice for that platform, is Java. Indeed, we've implemented an Android version of the Octopus UI *viewer*. These kind of devices are ubiquitous and have many interesting capabilities (GPS, touchscreen, accelerometer). The inclusion of Android supported devices to a distributed operating system, such as Octopus OS, seems promising.

Some notable differencies between the Android programming environment and JavaSE are: [12]

API The class library of the Android platform is a subset of the Apache Harmony Java implementation, which is different from the JavaSE.

Virtual Machine There is no JavaRE version for the Android OS. Davlik is a register-based[5] java virtual machine, specialized specifically for the platform. The Android SDK offers the tools to compile the Java code to Dalvik executables.

Graphics API Android does not use the Abstract Window Toolkit nor the Swing library. User Interface is built using View objects. Android uses a framework similar to Swing based around Views rather than JComponents.

2.3 Synthetic file systems

A synthetic file system [5] is a hierarchical interface to non-file objects that appear as if they were regular files in the tree of the disk-based filesystem. These non-file objects may be accessed with the same system calls or utility programs, as regular files and directories. The advantage of synthetic file systems is that, well known file system semantics can be reused for a universal and easily implementable approach to interprocess communication. Clients can use such file systems to perform simple file operations on its nodes, and do not bother with complex message encoding and passing methods and other aspects of protocol engineering. For most operations, common file utilities can be used, so even scripting is quite easy. The figure 2.1 clarifies further the idea behind the synthetic file systems.

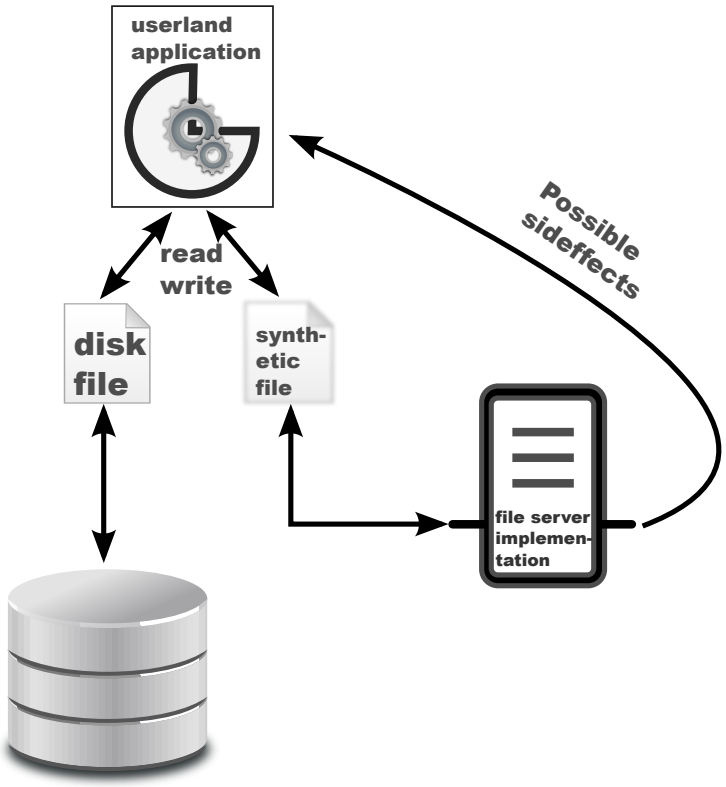
2.3.1 Styx

Styx [13] is a synthetic filesystem protocol implementation developed by the Bell Labs for the Inferno OS, which is a clone of the 9P protocol de-

[4] Android is a Linux-based operating system designed primarily for touchscreen mobile devices such as smartphones and tablet computers.

[5] As opposed to the stack-based JRE

Figure 2.1: An application operates on a synthetic file as if it is a disk file, effectively communicating with a synthetic file system server.



veloped for the Plan9 OS. The central idea behind the Styx protocol is to encode file operations invoked by a client upon a file system, into messages. These messages can be passed from one process to another or transmitted over the network, which enables transparent access to remote file systems. The representation of computing resources as a form of file system solves many difficulties of making that resource available across the network, because it allows uniform and transparent access.

Styx provides a view of a hierarchical, tree-shaped file system *namespace*, together with access information about the files (permissions, sizes, dates) and the means to read and write the files. Its users, that is, the people who write application programs, don't see the protocol itself, instead they see files that they read and write, and that provide information or change information.

A Styx communication involves two entities. A *server*, who provides a functionality through a synthetic file tree interface, and a *client* that exploits that functionality by invoking file operations to the nodes of the tree. The server implementation can be within the system's kernel or implemented as a userland application. The protocol defines 13 types of messages which are used for

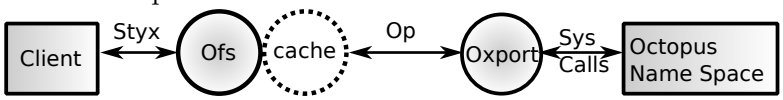
- Starting communication (attaching to a file system)
- Navigating the file system (that is, specifying and gaining a handle for a named file)
- Reading and writing a file
- Performing file status inquiries and changes

One of the protocols employed by the Octopus system is Styx and luckily there is a Java implementation of the protocol too. JStyx, as it is named, is used by the implementation of this project in order to be able to communicate with the Octopus PC.

2.3.2 Op

In the Octopus system, everything exported by the terminal is mounted as a file system through the network. The responsiveness of the system heavily depends from the file system protocol's performance. The Styx protocol, requires many message transmissions even for simple tasks. As a result, it does not behave well when high latency links occur because of the RPC round-trip time accumulation [14]. On the other hand Op [15], the *Octopus file system protocol*, is optimized for high latency communication links by batching multiple RPCs.

Figure 2.2: Oxport and Ofs are combined together in order to speak with a Styx client with Op



The protocol is accompanied by a number of tools that can be used to interconnect different Styx islands in a transparent way, keeping all other software unaware of the new protocol. These tools are, *oxport* that speaks Op as a server to export the namespace where it runs and *ofs* that speaks Styx as a server and Op as a client. The figure 2.2 illustrates how these tools are combined together to achieve the described functionality.

Chapter 3

Octopus OS

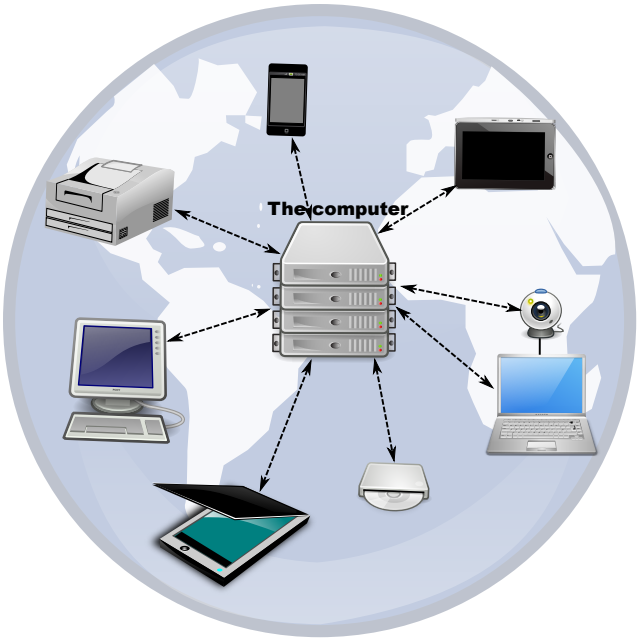
Nowadays a common user owns a number of computing devices which are possibly interconnected. The user is expected to manage a decentralized, uncoordinated, heterogeneous and highly dynamic environment. The net effect of trying to utilize all this devices is that the user acts as a controller who indicates explicitly at every use case scenario which computer they want to use for each thing done. This fact emerges from the assumption that all computers are equal. However if the centralization principle [2] is applied to the distributed environment, i.e by declaring a specific computer as *the central computer*, the management issues can be handled by that central computer freeing the user from that burden. The figures 3.1 and 3.2 illustrates the Octopus OS architecture.

The *Octopus* is a system, designed based on the centralization principle, which aims to provide a supporting platform to build distributed smart spaces and to provide pervasive applications that could be reached from anywhere. In the Octopus, there is a single dedicated computer per user, *the computer*, that executes all user programs independently of the user's location. A variety of software and hardware *resources* can be attached and exploited from applications running at *the computer*. Such devices can be highly heterogeneous, distributed, mobile and can be switched on and off at any time. On the other hand, the computer is a single, central, homogeneous system where all the applications run. This means that the interfaces between resources and the central part of the Octopus must be of a high-level of abstraction.

3.1 UpperWare architecture

UpperWare [3, 4] is a system architecture that permits to abstract and export the computing resources and ultimately to provide a synthetic file system interface for managing them. Such resources can be hardware de-

Figure 3.1: Various terminals connected to the central PC



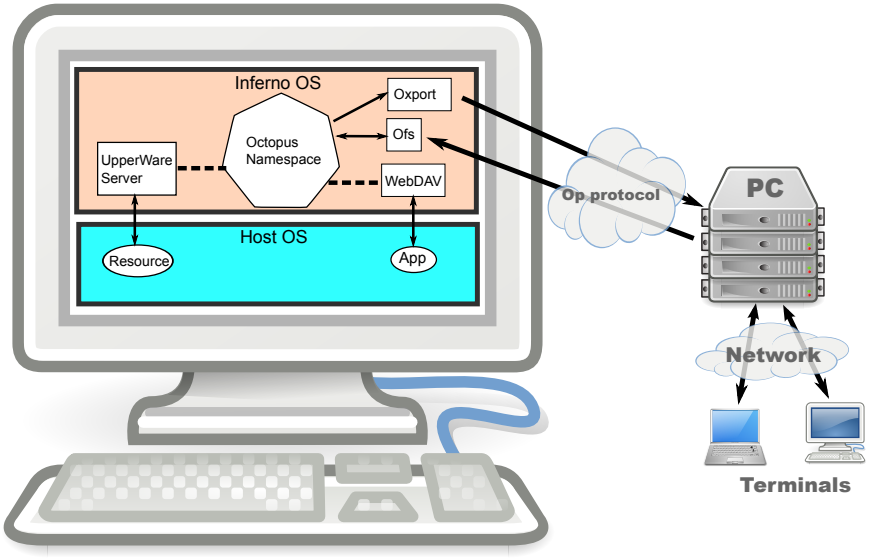
vices, applications, raw data and in general anything meaningful that can be wrapped reasonably by a file system interface. The UpperWare approach promotes the notion of a *virtual computer* composed by a variety of computing resources which is a crucial step towards making pervasive computing a reality. Octopus is build entirely out of UpperWare [16]. The relationship between the two is depicted in the figure 3.2

A piece of UpperWare providing a particular resource is called "UpperWare driver". There are two types of resource drivers.

- Passive UpperWare resource drivers process or generate data without any user interaction. They simply act as data sources or data sinks. The printer resource is an example of data sink because we provide to it data for printing and nothing else. The camera is an example of data source, i.e it provides a photo when prompted and aside from the initial configuration there is no other action required.

- Active UpperWare resource drivers are more complicated to develop because they have to support some sort of interactive control mechanism (e.g. popup dialog session). The user has to respond to inquiries according to the desired outcome. In practice, such interfaces have to be tailored for each particular resource because there are a number of ways of abstracting this kind of resources. An example of active resource could be a text editor which requires the constant intervention

Figure 3.2: An Octopus terminal that wraps, a filesystem interface, around a resource of the Host system and exports it to the Octopus global namespace *and* to the underlying Os through the WebDAV protocol.



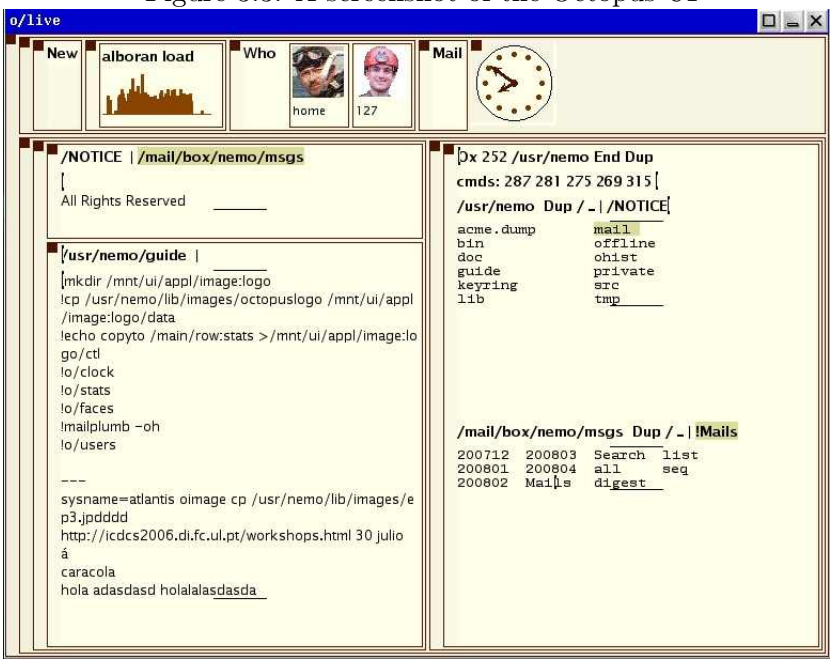
of the user while opening files, editing and copying them to disk.

In the Octopus system every machine with resources of interest runs some UpperWare and exports its resources to the PC through an appropriately designed synthetic file system. The PC provides a per-user global namespace that is shared by all machines of interest for the user. The namespace aggregates all resources and keeps them organized as hierarchy of files. Of course they are not files: they may be printers, applications, tools, data, etc. This makes application programming for Octopus very easy since they rely on UpperWare, that is, the only requirement is to be able to manipulate files.

3.2 Omero, a filesystem based window system

Omero [17] is the window system of the Octopus OS. The figure 3.3 presents a screenshot of the GUI of the system. Surprisingly, it does not draw and does not interact with the user, as this functionality is delegated to the *viewer* component of the system, named Olive. Omero implements a file tree that represents a hierarchy of graphical components, known as *panels*. This design approach offers many subtle advantages compared to the traditional GUI systems. For one, it's simple enough to not require a sophisticated programming interface (since it's based on file operations only). The added

Figure 3.3: A screenshot of the Octopus UI



value though is the seamless UI distribution and replication. The figure 3.4 presents a basic diagram of the windowing system architecture.

At the root of the Omero hierarchy, there are directories representing virtual screens, a directory named *appl* that contains application related panels and a file, named olive, used for delivering UI update events. There are three general categories of panels: rows, columns, and atoms. Rows and columns are components employed for layout purposes, whereas atoms are graphical components such as text, images, buttons etc. Every Panel has it's corresponding directory, which is served by Omero. Each directory contains a file named *ctl* and optionally additional files describing further a particular panel.

An Omero Panel can be on of the following types [18]:

row A container panel arranging children in a row.

col A container panel arranging children in a table.

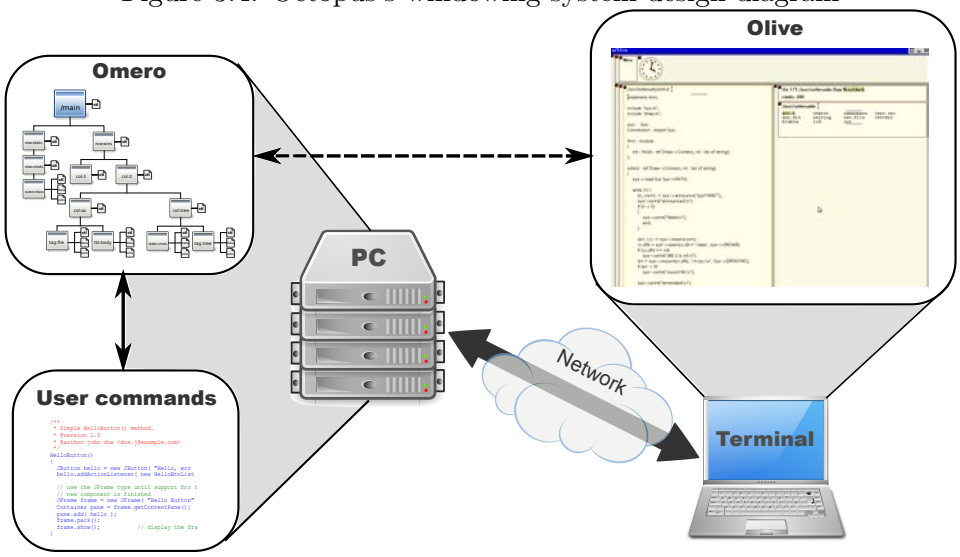
image An image in Plan 9 format.

text An editable text panel.

tbl An editable text panel that insists on tabulating the words contained.

label A single line (small) read–only text panel.

Figure 3.4: Octopus's windowing system design diagram



button A single line (small) read–only text panel customized to behave as a button.

tag A single line editable text panel. Usually to inform the user of sibling panels and to provide a place to type some text.

gauge A meter to show a value between 0 and 100.

slider An editable meter to show a value between 0 and 100 and let the user adjust it.

page An image in Plan 9 format supporting paning. To view large images.

draw A vector graphics device. Used to draw geometrical figures.

The type of a given panel element is denoted by prepending the name of the respective directory with the type. It has to be clarified, sooner rather than later, that Panels that have the *tag* attribute contain a small rectangular area to the upper left corner that is used to move panels with mouse and issue other commands. This one should not be confused with the *tag* Panel which is entirely different thing.

The filesystem based architecture employed by Omero, offers a number of interesting functionalities.

distribution The user can move any part of an application's interface to the terminal of his choice simply by moving its corresponding filesystem

hierarchy branch to the directory responsible for the (virtual) screen assigned to that device. For example you may simply move the "next slide" button to your mobile phone and use it like a remote control without bothering the application. The UI can merge too by moving the panel directories under the same *screen* file system hierarchy.

replication One can just copy a number of Panel directories, to multiple screen hierarchies, effectively replicating the corresponding UI to multiple displays. Omero takes care of synchronizing the editions between replicas. This functionality is extremely useful for creating collaboratives applications without bothering with complex and tedious tasks like synchronization over the network.

general purpose tools Legacy tools like find, grep, tar, cron, etc are reusable in the Omero environment. One can just save the whole screen in a package with tar and load it afterwords by unpacking it in a screen directory. When traditional window systems have to implement a "search window by title" functionality, an Octopus user can just use the *find* tool written a couple of decades ago. A crontab job with less than 10 lines of code can be assigned to hide windows that are idle for considerable time whereas developing the same functionality for a traditional window system would require non-negligible programming effort.

simple API In order to develop a graphical application for Octopus, the programmer has to use a filesystem interface which is well supported from programming languages and well understood by programmers. Compared to the traditional GUI application development where the programmer has to learn and use sophisticated libraries, the filesystem based API much more simpler and cleaner. Although it should be noted that some flexibility is sacrificed due to the fact that, one cannot create custom panels.

3.3 Olive, the Omero viewer

Olive [17] is a viewer that permits the user to interact with Omero. It generates graphical components on the monitor to display panels according to the filesystem hierarchy supplied by Omero. It accepts mouse and keyboard input to operate on the panels by mapping the user's actions to filesystem operations. O/live is the only program that knows how to draw, how to interact with the mouse and the keyboard, and how to implement graphically a particular panel type. The interaction that occurs is based on high level file-based operations. In essence Olive is a *client* application for the Omero file-server. The figure 3.5 presents the mapping between the filesystem and the visual components that Olive generates.

Figure 3.5: Mapping of a filesystem tree to graphical components.

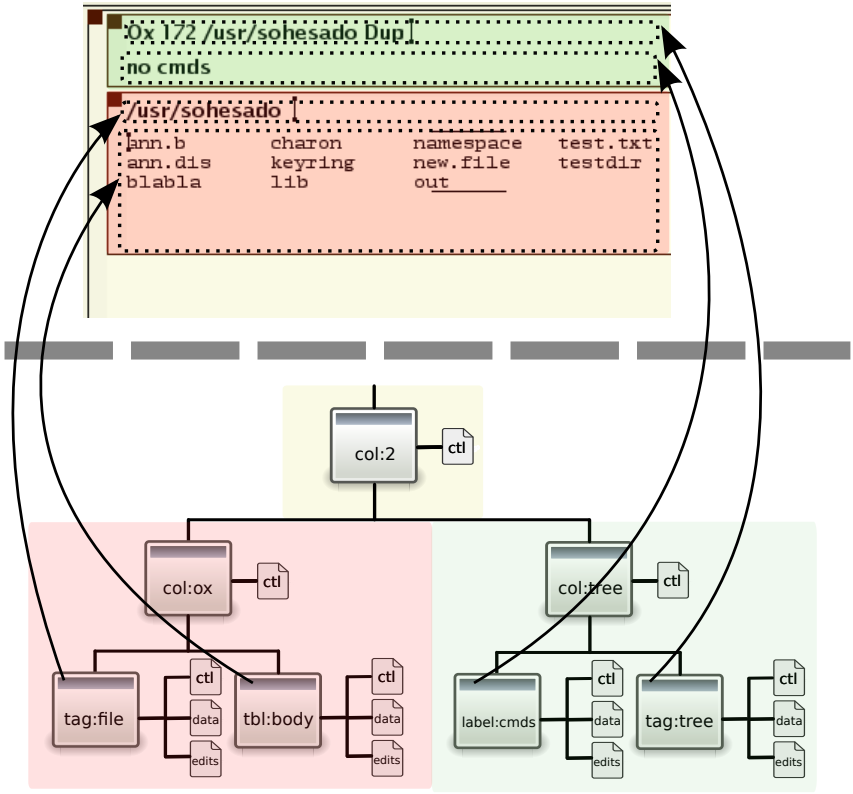


Figure 3.6: A popup menu

While on a tag or margin, the mouse can be used as follows:

Button 1 A drag on a tag moves the tagged panel. On rows and columns a small vertical or horizontal drag makes the container behave like a column or a row. A single click resizes the panel according to the following mouse actions: Another single click (i.e., a double click) adjusts the size automatically and a drag changes the size of the panel in proportion to the destination of the drag. recomputes the layout for the tagged panel.

Button 2 A single click on a tag maximizes the panel, by hiding its siblings on the outer row or column containing it. If the panel is already maximized, a single click shows all siblings, undoing the effect.

Button 3 A click raises a menu with panel operations. The menu items are selected by moving the mouse the direction of the name. The menu with its items is illustrated in the figure 3.6 . A drag can be used to adjust the size of the panel.

The keyboard, aside from typing texts into the panels, function keys 1, 2 and 3 act as the respective mouse buttons for the cases that the mouse has not 3 buttons.

3.4 Ox, the Octopus shell

Ox [18] is an octopus application that implements a shell to browse the file system, edit, and execute commands. Ox can be used in an implicit and an explicit manner. By invoking tag and panel operations which are offered by the window systems viewer directly, the user implicitly handles Ox since these operations are mapped by the underlaying system to o/x operations. This kind of usage scenario is the common case, however o/x supports its own command language consisting of builtin commands, Sam [1] commands, host [2] commands, and inferno commands. The user is able to invoke these commands by typing them explicitly.

[1] A text editor for programmers developed by Rob Pike
[2] the OS hosting the inferno instance

Host commands are commands executed to the system, hosting the Inferno (e.g Linux).

Inferno commands are commands executed by the Inferno system underlying Octopus.

Sam commands editor commands similar to the Sam editor command language.

Builtin commands refer to the respective manual page ox(1).

As an example [18], consider the use case scenario in which the user desires to save the state of the GUI permanently to the disk with the intention to recover the session in the future. With Ox he can simply

Save the session:

```
% cd /mnt/ui/appl
% tar c col:ox.* >/tmp/oxui.tar
```

Load the session:

```
% cd /mnt/ui/appl
% tar x </tmp/oxui.tar
% ox -l /mnt/ui/appl/col:ox.*
```

In order to achieve the same effect in a traditional windowing system, it would require non-negligible number of lines of code which will further complicate the system.

Chapter 4

Java Octopus Terminal

In a sense, Octopus is an aggregation of UpperWares, i.e. resources wrapped with a filesystem interface and exported to a global namespace managed by the PC. By employing this design, we abstract the details of heterogeneous components with the intention to import them to a homogeneous, centralized system. The key software system technologies used to achieve this effect are Inferno and Styx/Op. The former for abstracting the operating systems that Inferno supports, and the later to serve the role of the synthetic filesystem protocol. A great number of resources can be incorporating in the Octopus pervasive environment but not too many and since the system's effectiveness increases proportionally to the number of compatible resources, it is reasonable to try to build more UpperWares in order to enhance the system's usefulness.

In order to broaden the spectrum of candidate Octopus resources we started to build a Java Octopus terminal. Practically we traded Dis with JavaRE and Styx/OP with JStyx. The rationale behind this decision is that there are many Java-based devices that can't host an Inferno instance hence can't be included in the Octopus environment. Also there are platforms that can host Inferno but it is not practical from the user's point of view (e.g Android). The ultimate goal is to incorporate Java-based terminals along with the traditional Inferno-based, gracefully .

There are a number of advantages of a Java-based approach compared to the original Inferno-based.

- JavaRE is more ubiquitous compared to Dis.
- Installation and execution is easy.
- Homogeneous access to resources across platforms.

- To be able to exploit the native GUI support and look&feel of the platform.

The development of a Java terminal's goal is to enrich the Octopopus "ecosystem", not to replace the Inferno terminal. Whenever the user has to choose between the two, the original terminal is the way to go. Having said that, the number of mobile platforms supporting Java is an "audience" that should not be easily ignored.

4.1 JOlive

JOlive[1]s Olive's Java counterpart which implements the *viewer* component of the window system. The figure 4.1 provides an overview of the two JOlive implementations compared to the original *viewer*. In essence, it is a terminal that offers a display resource to an Octopus system. In an abstract sense JOlive by utilizing file operations, "reads" the state of the UI, "writes" modifications based on the interaction with it's user, and again "reads" updates of the UI state. At low level the communication between JOlive and Omero is based solely on Styx messages. The remote synthetic filesystem served by Omero is exported from the PC to the client that runs JOlive and based on this communication the visual part of the UI is shaped in order to map, the file operations, to graphical interface actions.

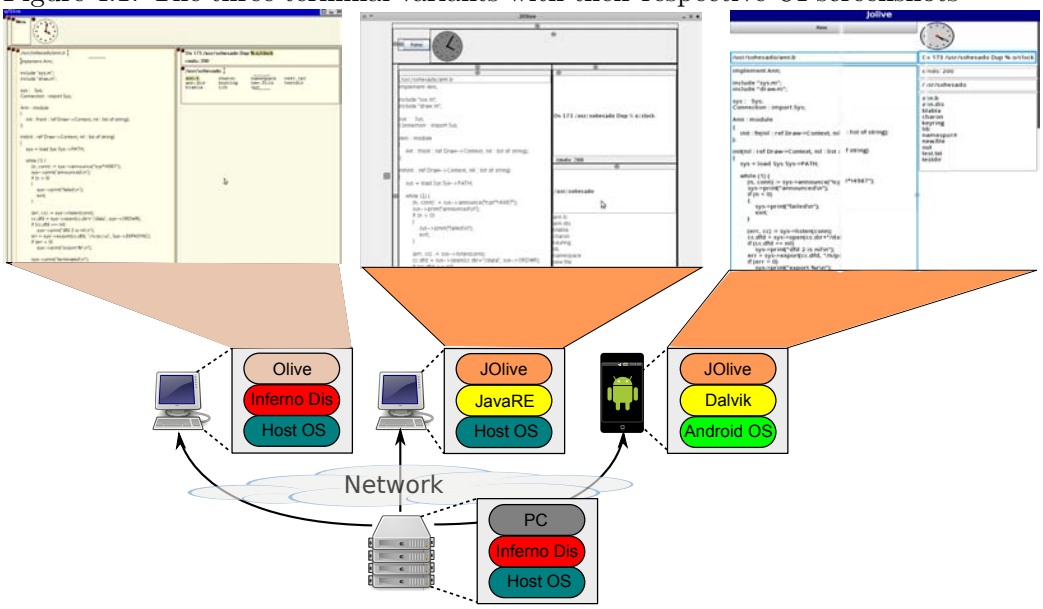
There is a bidirectional communication between Omero and JOlive. There is the *user action notification* and *UI update notification*. The first occurs whenever the user interacts with the graphical user interface and the second whenever a change occurs to the UI remote file system, by an application or interaction through another viewer. These two data flow paths complete the action-effect feedback loop.

User action notification Every visible widget has its corresponding directory to the remote file system. So when a user invokes an action (e.g button click event) that is mapped to a certain file operation according to Omero's protocol. In this manner the file-server is notified for the user's actions which in turn updates the corresponding application.

UI update notification When the filesystem is updated, either by an application or viewer, Omero generates event messages which mirror that changes. The viewer retrieves these event messages by reading the ui/olive file and update their graphical structure.

[1] i

Figure 4.1: The three terminal variants with their respective UI screenshots

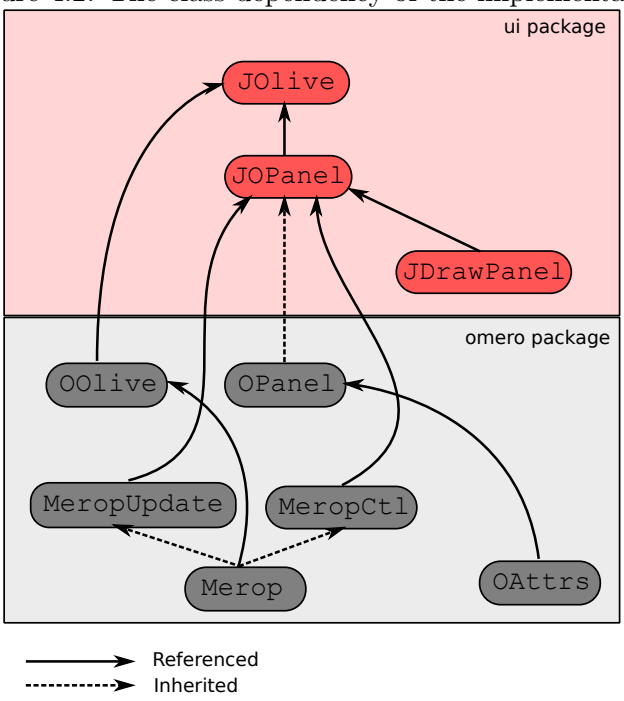


Both Android and JavaSE are implemented according to this design. The two datapaths are independent with each other and can be refined separately.

4.2 Desktop version

The Desktop version is implemented in Java Standard Edition version 5 based on JStyx [19] library. The graphical part is implemented based on Java/Swing API which is a *certified* package, guaranteed to be available on all feature Java updates. The implementation consists two packages, *omero* and ui. The former contains the back-end implementation, responsible for communicating with Omero, the later implements the visual and user control components. The relationship between them, is depicted in the figure 4.2. The coupling between the two components is is kept as minimal as possible which allows the modification of each component in a modular fashion, as well as to selectively reuse those components in other implementations. For example we can keep the UI package intact, and modify the back-end to communicate with JOp (a prospective Op Java implementation) instead with JStyx. Perhaps more likely, we can reuse the back-end and change the visual part of the implementation. Actually this is the case of the Android implementation, as it will be clarified in the respective section.

Figure 4.2: The class dependency of the implementation



4.2.1 Omero package

This part of the implementation is the back-end of JOlive and contains the synthetic filesystem dependent parts. The package consists of the following classes.

OPanel This is the most important class of the package which encapsulates the filesystem details of an Octopus panel. The means of communicating with Omero is implemented within this class, along with some attribute details. The back-end of the *user action notification* is implemented here. After initializing an OPanel object, it can be utilized to invoke *ctl* commands, get or set *data* and read the Panel specific attributes. The most part of the communication with Omero is implemented here and any future extensions on that respect probably will occur in this class.

OOlive This is the second most important class of the package because it encapsulates functionality offered by the *olive* file served by Omero. It is delegated to implement the back-end of the *UI update notification* by listening for update events and demultiplexing them in based on the "Panel" objects contained in JOlive. At first, it decodes the event messages and it passes them to the update function of the respective JOPanel object. In order to do so, a global hash table is maintained

which maps Panel paths to JOPanel objects. Since every event message contains the path of the respective Panel, said demultiplexing is possible. Additionally, every *event message* is self contained and fully describes the required changes to be applied, so there is no need for further actions.

Merop, MeropCtl, MeropUpdate These three classes encapsulate the information related to the event messages generated by Omero. Merop is the parent class of the other two, which refine's further the objects based on the message type. These three classes implement the message decoding, encoding and storage functions. Such objects are passed by OOlive to the respective JOPanel update functions.

OAttrs, OUtils OAttrs encapsulates the attributes that a Panel may have and OUtils contains debugging functions.

4.2.2 ui package

This package implements the front-end of JOlive and contains the UI toolkit dependent parts. The package consists of the following classes.

JOPanel This class is a direct descendant of OPanel and additionally contains the graphical implementation of the Panels. Specifically, it has a reference to a *Swing* component which is initialized based on the type of the Panel and the means to update and control that component. Every JOPanel, upon initialization, register themselves to the OOlive's hashtable and create a swing component based on the Panel type that they represent. This is the most complex class of the implementation since it is the "glue" component that combines the back-end with the front-end. Every JOPanel object has the means to display itself to the screen, to invoke Omero commands and to update itself based on the event messages supplied to its update method.

JDrawPanel Is a customized JPanel widget, employed to represent the Omero's draw panel. It contains draw instructions and is able to transform them to accommodate to the Swing specific instructions. This class of objects are generated by JOPanel objects that represent a draw Panel.

JOlive Is the main class that initializes the window and initiates a connection with the Octopus PC.

You can get the source code from the github location at
`https://github.com/sohesado/JOlive`
. If you want to clone the repository, use
`https://github.com/sohesado/JOlive.git`
git address.

4.3 Android version

Given the popularity of Android devices, and the fact that there is no practical[2] way to run an Octopus terminal on such devices, is a valuable addition to the Octopus "ecosystem". By setting up JOlive on an Android device, the user is able to interact with an Octopus installation through network. One other important benefit is the fact that the UI is implemented with the native Android graphical widgets which offers an integrated look & feel experience. In a more general sense it enhances the versatile, loosely coupled nature of Octopus's UI, meaning that one is able to accommodate the front-end to his liking. More importantly he is able to adapt to the terminal device's capabilities, nicely exploiting the support of Android on many different platforms (thus being relieved from having to deal with numerous low-level UI issues). As it is stressed out already, the fact that Omero offers the flexibility to implement graphical *viewers* separately, is a major advantage since there is the opportunity to provide "native" GUI for a given terminal device.

The first requirement that has to be satisfied is the synthetic filesystem based communication capability. With some tinkering, JStyx has been ported successfully to the Android OS, which enabled the development of JOlive. Luckily, the Desktop version was implemented based on JStyx too so the *Omero* package was reused almost unchanged. Unfortunately the Android API does not support AWT/Swing so the UI package had to be reimplemented based on the Android graphical toolkit.

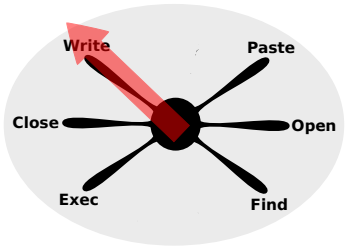
4.3.1 com.jolive.Omero

As it was stated above, the Omero package required only some minor changes while ported to Android. This change was based on a certain technical restriction that the Android framework imposes. Specifically it demands that every *View* object is allowed to be updated from the same thread that created it. Because the OOlive component has to run on a dedicated thread since it blocks for events generated by Omero, it cannot invoke graphical updates because the GUI initialization occurs in the main thread. In order to work around this restriction the OOlive update mechanism had to be modified.

The update problem was solved by utilizing a handler object generated by the main activity thread. Specifically when the application is initiated, a message handler is created that asynchronously passes the event messages to the respective JOPanel objects. The reference of this handler is passed

[2]Although John Floren did an amazing work on porting inferno to the Android OS, many users may not be willing to abandon the Java stack [20]

Figure 4.3: The Red arrow indicates the finger's path in order to issue the Write command



to the OOlive component when it is initialized, which now is able to pass any event message it receives to that handler. Because that handler is part of the main thread, it can legitimately update the UI components.

4.3.2 com.jolive.ui

The com.jolive.ui package contains the following classes. verbatim list of classes.

JOliveActivity, JOPanel, JDrawPanel These classes implement the equivalent functionality of their counterparts for the Desktop version. Aside the refactoring required to accommodate the differences between the Swing API and the android API there isn't a notable difference.

ActionSwipeListener, TextLongClickListener Because it isn't elegant to use long dropdown menus for small touchscreen devices, we had to implement a more efficient method of invoking commands. Fortunately, the original Olive support such an elegant method 3.6 So we created a similar one that work by substituting the mouse with touchscreen. Practically the user has to put the finger on the center of the widget and swipe it in the direction of the command he wants to issue. Figure 4.3 shows the widget. Technically, 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.

PullAppListener This class implement the application pull functionality explained in the next subsection.

ConfigureLog4j This class initializes the event logging mechanism built in JStyx. It was a technical requirement in order for the JStyx port to succeed and it is not used from JOlive.

You can get the source code from the github location at `https://github.com/sohesado/android-JOlive`.

If you want to clone the repository, use
`https://github.com/sohesado/android-JOlive.git`
git address.

4.3.3 Pull application's UI

The user may have numerous, many tens of open apps. But the user may wish to use the terminal to interact with just a few or perhaps even just one app at a time, which makes sense especially 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.

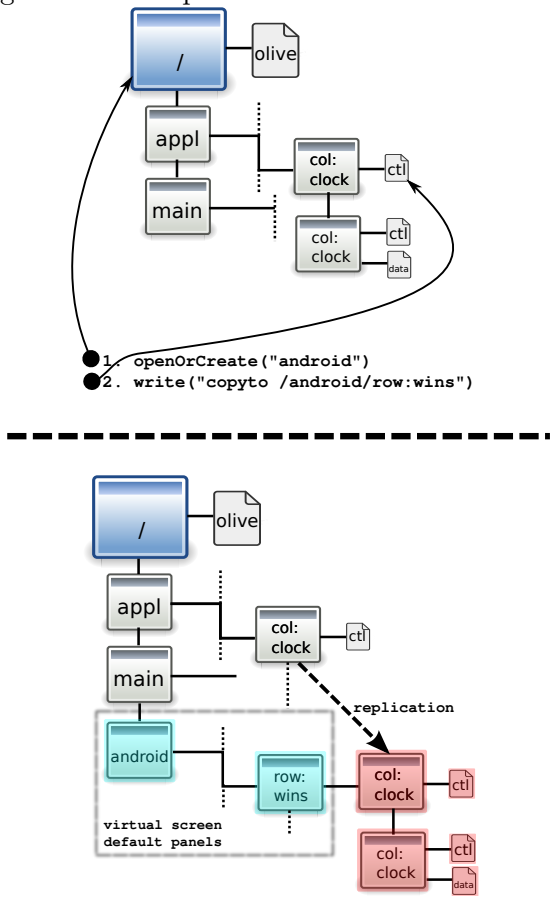
Consider the following motivating examples. I want to use my mobile phone to browse and add an appointment in my calendar that is displayed on the main screen, along with my other apps. Instead of scrolling through the numerous running application panels, I *pull* the calendar related panels to the mobile phone's screen. One other group of use case scenarios where the *pull* functionality would be useful is the opportunity to use the android mobile phone as a remote controller. For example, the user makes a presentation with a projector and desires to control the slides remotely. He can pull 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 the help the user to easily select panels, rendering the small screen usable, without bloating it with numerous Panels. The procedure can be break down to the following steps.

1. Push the menu button.
2. Select the Pull function. Pops up a checkbox list with the apps running at the PC.
3. Check the desired panels
4. Touch the Pull button. As a result you get on your screen only the selected panels.

In Figure 4.5 is illustrated the process of "pulling" the clock application.

Figure 4.4: The pull effect on the Omero tree



Under the Hood, JOlive initializes a new virtual screen by creating a directory at the Omero's root and consecutively issues *copyto* 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. The figure 4.4 shows the effect of the pull function has on the Omero UI tree. 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 with a mobile phone. With this option, the user with a couple of clicks achieves the result of textual commands that had to be passed manually to Ox, which isn't practical for a small screen device with a virtual (touchscreen) keyboard.

Figure 4.5: Step by Step Pull App example

Chapter 5

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 functionality that it lacks, in order to be considered a complete terminal, is to expose various resources provided by the underlying system, to the PC. The following sections describe briefly some interesting resources and other terminal related extensions.

5.1 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. For example if the Android terminal is able to export its GPS coordinates through a passive UpperWare resource driver the global Octopus namespace, it will enhance the smart space characteristics of the system. The user can easily "calibrate" his working environment e.g. by registering indirectly the coordinates of a number Desktop computers (he puts his phone close to the Desktop, and *copies* the file representing the coordinates of the phone to a permanent registry). The system can exploit this information and offer event triggering API based on GPS coordinates.

As an example, consider a system administration use case scenario in which the admin desires to physically inspect a number of terminals. He can script a hook procedure that generates information related to a certain terminal when its GPS coordinates match with phones (assuming the admin will carry his mobile device with him). For his convenience he can project that info to his mobile device too (e.g. tablet PC).

5.2 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 a useful tool on its own. This being said, the Java terminal can be really benefit from an Op implementation especially the Android version when it will communicate over the cellular network (e.g. 3G data transfer network).

5.3 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 the hardware device. One possible way to to this is to combine a user defined touchscreen gesture with the hardware ID of the device and generate a hash value that authenticates the user's credentials. The hash 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. This resource can be wrapped with a passive UpperWare driver and exploited by other system resources that require a more secure authentication mechanism.

5.4 Remote Voice commander

The Octopus implementation experiments with the notion of integrating voice support to the system. If it succeeds in providing a novel voice command/feedback control loop in an integrated fashion, it will be an undisputed advantage compared other pervasive environment implementations. The smartphones have decent voice recording capabilities. A possible interesting 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 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 it is usefull to communicate easily (i.e. voice commande) with the system while the user is on the move.

5.5 Conclusion

During this projects I encountered many interesting technologies like Inferno OS and of course the Octopus system. Also I familiarized myself with abstract notions like the UpperWare architecture and the concept of centralizing everything, with the goal to solve a number of distributed environment problems. The development effort presented its challenges too. At first I had to understand the system in great depth, sometimes I was forced to read the Octopus implementation's source code too, but in the end with my professor's valuable support we managed to implement a Java Octopus terminal prototype.

Bibliography

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

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

[5] Synthetic file systems, wikipedia article. `http://en.wikipedia.org/wiki/Synthetic_file_system`.

[6] 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.

[7] Inferno OS, wikipedia article. `http://en.wikipedia.org/wiki/Inferno_os`.

[8] Inferno OS, vita nuova homepage. `http://www.vitanuova.com/inferno`.

[9] Limbo, vita nuova homepage. `http://www.vitanuova.com/inferno/limbo.html`.

[10] JavaSE, wikipedia article. `http://en.wikipedia.org/wiki/JavaSE`.

[11] C. F. Yurkoski, L. R. Rau, and B. K. Ellis. Using inferno:::tm::: to execute java:::tm::: on small devices. In Frank Mueller and Azer Bestavros, editors, *Languages, Compilers, and Tools for Embedded Systems, ACM SIGPLAN Workshop LCTES 98, Montreal, Canada, June 1998, Proceedings*, volume 1474 of *Lecture Notes in Computer Science*, pages 108–118. Springer, 1998.

[12] Android compared to javaSE, wikipedia article. http://en.wikipedia.org/wiki/Comparison_of_Java_and_Android_API.

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

[14] 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.

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

[16] 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.

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

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

[19] Jstyx is a pure-java implementation of the styx protocol. http://www.resc.rdg.ac.uk/jstyx/.

[20] Inferno for android phones. https://bitbucket.org/floren/inferno/wiki/Home.

Chapter 6

Thesis preview in Greek

Υποστήριξη σε java της γραφικής διεπαφής χρήστη του κατανεμημένου λειτουργικού συστήματος Octopus

Εισαγωγή

Ο σύγχρονος χρήστης υπολογιστών στην εποχή μας έχει στην κατοχή του έναν αριθμό από υπολογιστές (desktop, laptop, tablet, smartphone), οι οποίοι είναι εν δυνάμει συνδεδεμένοι μεταξύ τους μέσω του διαδικτύου. Ο χρήστης για να καλύψει τις ανάγκες του, αντιμετωπίζει ένα αποκεντρωμένο, ασυντόνιστο και ετερογενές υπολογιστικό περιβάλλον. Είναι αναγκασμένος κάθε φορά να επιλέγει ρητά ποιόν υπολογιστή θέλει να χρησιμοποιήσει, και όταν επιθυμεί να πετύχει κάποια συνεργατική λειτουργικότητα θα πρέπει να διαχειριστεί τον συντονισμό μεταξύ των συσκευών, ο ίδιος. Πολλά απ'αυτά τα προβλήματα προκύπτουν απ'το γεγονός ότι αντιμετωπίζουμε όλους τους υπολογιστές ως ισάξιους.

Το σύστημα Octopus αποσκοπεί στο να λύσει αυτά το προβλήματα προσφέροντας ένα ομοιογενές, απανταχού παρών προσωπικό διάχυτο υπολογιστικό περιβάλλον, με το να κεντρικοποιήσει όλους τους πόρους σε έναν υπολογιστή. Επομένως ανακηρύσσουμε έναν υπολογιστή ως το κεντρικό PC, και συγκεντρώνουμε όλους του πόρους σε αυτό.

Αυτή η προσέγγιση έχει κάποια σημαντικά πλεονεκτήματα:

- Ο συντονισμός είναι εύκολος, αφού όλοι οι πόροι έχουν συγκεντρωθεί σε έναν υπολογιστή.
- Παρέχετε η δυνατότητα ομοιογενούς πρόσβασης σε εν γένει ετερογενείς πόρους.
- Πρόσβαση μέσω Internet σε όλους τους πόρους του συστήματος με διαφανές τρόπο.

Υποστηρικτικές τεχνολογίες λογισμικού

Τα πρωτόκολλα συνθετικών συστημάτων αρχείων είναι μια ιεραρχική διεπαφή αντικειμένων, που φαίνονται σαν αρχεία, αλλά δεν είναι. Το κοινό στοιχείο που υπάρχει μεταξύ τους είναι η διεπαφή χρήσης τους. Το octopus χρησιμοποιεί δύο τέτοια πρωτόκολλα, το Styx, και το Op (Octopus protocol). Το πρώτο χρησιμοποιείτε κυρίως σε περιβάλλον τοπικού δικτύου ενώ το δεύτερο σε περιπτώσεις που υπάρχει μεγάλη καθυστέρηση διαδικτυακής επικοινωνίας.

Το λειτουργικό σύστημα Inferno OS προσφέρει ένα crossplatform περιβάλλον ανάπτυξης κατανεμημένων εφαρμογών με την χρήση του πρωτοκόλλου Styx, στην γλώσσα προγραμματισμού Limbo. Το σύστημα Octopus έχει αναπτυχθεί αξιοποιώντας τα χαρακτηριστικά που προσφέρονται απ'αυτήν την πλατφόρμα, με στόχο να παρέχει στους χρήστες του ένα προσωπικό διάχυτο υπολογιστικό περιβάλλον.

Η αρχιτεκτονική του Octopus OS

Το Octopus βασίζεται στην UpperWare αρχιτεκτονική. Το κεντρικό νόημα αυτής της προσέγγισης είναι να κωδικοποιηθούν οι διάφοροι υπολογιστικοί πόροι των συστημάτων που ανήκουν στο περιβάλλον του, με διεπαφή συνθετικών αρχείων συστήματος. Κατ' αυτόν τον τρόπο όλοι οι πόροι αποκτούν ομοιογενείς διεπαφή χρήσης και συγκεντρώνονται σε ένα ενιαίο ιεραρχικό σύνολο.

Το παραθυρικό περιβάλλον

Το σύστημα παραθύρων, όπως και οι υπόλοιποι πόροι του συστήματος, είναι κωδικοποιημένο με ένα συνθετικό δέντρο αρχείων. Αυτό εξυπηρετείτε απ'το Omero το οποίο παρέχει την βάση για την δημιουργία γραφικών διεπαφών χρήσης εφαρμογών που εκτελούνται στο Octopus. Η ιεραρχία των αρχείων αυτών αντιστοιχείται σε γραφική αναπαράσταση από ένα άλλο κομμάτι λογισμικού το οποίο ονομάζεται Olive. Αυτό εκτελείτε (κατα κανόνα) στα τερματικά στοιχεία του συστήματος ενώ το Omero εκτελείτε στο κεντρικό PC.

Octopus τερματικό σε Java

Στα πλαίσια αυτής της εργασίας έχει αναπτυχθεί το JOlive το οποίο είναι ένα Omero viewer υλοποιημένο σε Java. Κατ' αντιστοιχία με το Olive, απεικονίζει τον κάθε φάκελο (μαζί με τα αρχεία του) με το γραφικό στοιχείο που του αντιστοιχεί. Για παράδειγμα ένας φάκελος με τίτλο button:someapp αντιστοιχίζετε σε ένα κουμπί το οποίο ανήκει στην εφαρμογή someapp.

Ως επέκταση της παραπάνω υλοποίησης μεταφέραμε το JOlive στην πλατφόρμα Android, με τις όποιες απαιτούμενες αλλαγές για να επιτευχθεί αυτό. Το τελικό αποτέλεσμα είναι να ενσωματωθούν οι συσκευές Android στο "οικοσύστημα" των εν δυνάμει πόρων του συστήματος Octopus.

Επεκτείναμε την έκδοση του Android με κάποιες επιπλέον λειτουργίες οι οποίες διευκολύνουν την χρήση συσκευών με περιορισμένες δυνατότητες προβολής και έλεγχου. Όσον αφορά τον έλεγχο υλοποιήσαμε ένα γραφικό εργαλείο που μας δίνει την δυνατότητα να επιλέξουμε μια εντολή από σύνολο επιλογών, μια απλή κίνηση στο touchscreen. Όσον αφορά τις δυνατότητες προβολής η λειτουργία Pull App που υλοποιήσαμε, δίνει την δυνατότητα επιλογής ενός υποσυνόλου του
συνολικού γραφικού περιβάλλοντος, αφού αυτές οι συσκευές συνήθως έχουν οθόνες περιορισμένων διαστάσεων.

Γενικά, η συνεισφορά μας στο σύστημα Octopus ήταν να επεκτείνουμε το εύρος των εν δυνάμει πόρων, με την προσθήκη της Java πλατφόρμας στις υποψήφιες υποστηρικτές τεχνολογίες.