Omero: Ubiquitous User Interfaces in the Plan B Operating System*

Francisco J. Ballesteros Gorka Guardiola Katia Leal
Enrique Soriano
Laboratorio de Sistemas Universidad Rey Juan Carlos
Madrid, Spain.
{nemo,paurea,kleal,esoriano}@lsub.org

Abstract

It is difficult to build user interfaces that must be distributed over a set of dynamic and heterogeneous I/O devices. This difficulty increases when we want to split, merge, replicate, and relocate the UI across a set of heterogeneous devices, without the application intervention. Furthermore, using generic tools, e.g. to search for UI components or to save/restore them, is usually not feasible. We follow a novel approach for building UIs that overcomes these problems: Using distributed file systems that export widgets to applications. In this paper we describe Omero, a UI server built along this line for the Plan B Operating System.

1. Introduction

Ubiquitous computing environments provide the user with multiple displays, pointing devices, and keyboards. Therefore, it is desirable that user interfaces (UI) could work in this distributed and heterogeneous environment and take advantage of the distribution. There are many different toolkits, frameworks, and UI management systems (UIMS) for implementing UIs, e.g., [12, 10, 9, 13, 5]. The differences among them are wide yet we found similar problems while trying to build our smart space:

It is hard to simultaneously use different devices when applications or users require to distribute the UI among them. For example, a presentation viewer may want to deploy a control panel on a phone's display, a slide viewer in a large graphical display, and accept several audio commands.

It is hard to split and merge the UI and place different parts of it on whatever device is considered appropriate. Once programmed, we cannot split a given UI component into separate ones.

Replication of UI components is hard and requires collaboration from the application. It is desirable to be able to replicate UI elements without placing the burden on the application.

General purpose tools do not work on UI elements. This is a big problem for a smart environment, because it requires many programs to make it *smart*. For UI elements, tasks already accomplished by general purpose programs (e.g., searching with `find` or `grep`, or copying with `cp`) requires writing specific purpose software for the task and UI considered [17].

It is hard to consult and update information about the UI itself, e.g., to obtain the label for a button from a different program or to update the label to something else.

While constructing the Plan B OS [1], which supports our smart space, we have developed an architecture for building UIs that overcomes these limitations. Our approach is to implement and export UI components (i.e., widgets of a high-level of abstraction) by means of network file systems. The hierarchy of UI elements found in a UI is represented by a file hierarchy, following the ideas in [8, 16]. Graphical displays and other devices employed for UIs are supported by UI file servers that implement a set of widgets and permit their use through the file system interface.

As a result, the application can program and use its UI in the same way it uses regular files, and it can be mostly unaware of the actual set of devices used to deploy the UI. Furthermore, external programs can rely on the file interface to inspect and operate on existing UI components. Different devices are accessed by mounting their UI servers and using their file trees to build and use different UI components.

2. Omero

Omero is the Plan B window system and the User Interface service, it has been used in production for almost a year. The current implementation works both on Plan 9 [16] and Plan B [1], but the approach can be applied to any other system. A typical user employs multiple screens, serviced by different omeros, that may look like the one shown in figure 1. Unlike in other systems, omero implements both

*This work supported in part by spanish MCyT TIN-2004-07474-C02-02. E. Soriano also supported by FPI grant BES-2003-2942.

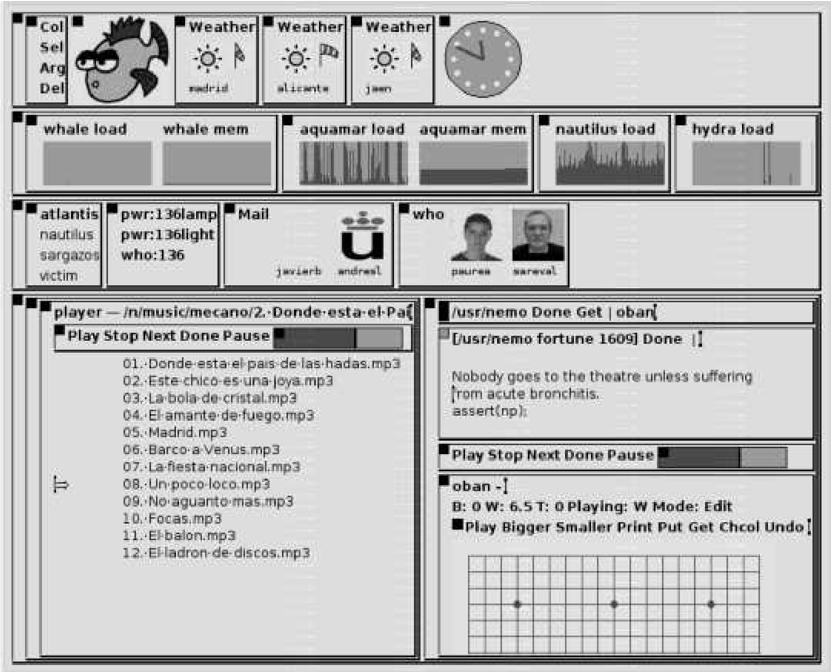

Figure 1. A typical Plan B screen, serviced by the `omero` UI service.

window management and the set of GUI components available. In this respect, it is both a window system or UIMS and a GUI toolkit. The user interacts with omero using any keyboard and pointing device available in the network. Applications interface with omero using the files it provides.

A screen handled by omero consists of a tree of UI elements known as panels. There are three kind of panels: *rows*, *columns*, and *atoms*. Atoms include text, images, gauges, and the like. All panels, including rows and columns, are considered the same by omero, like in Morphic [11]. They can be moved around, copied, pasted, hidden, deleted, and so on. For omero it does not matter if a panel is part of an application's UI, the entire UI, or a row/column created by the user to group other panels.

The graphical representation of panels in the screen corresponds to the file tree serviced by omero to its clients. For example, a screen that contains two rows has two corresponding files in its root directory. If the user moves one row within the other using the mouse, the same would happen to their respective files; and vice-versa.

The file tree is exported using a remote file system protocol, and can be mounted from anywhere in the network. Several devices supporting UIs can be used together by mounting their respective servers. Applications operate the service by mounting one or more omero's file trees on their name space and using the standard file operations.

The interaction with the mouse and the keyboard happens within omero, without the intervention of the application. The mode of interaction is very similar to that of Acme [15], which can be considered a direct ancestor for omero.

3. Widgets as files

Omero represents each panel by a directory that contains a `ctl` and a `data` file (see figure 2). Panels can be created and deleted by making and removing such directories. Once the user has created a directory, omero automatically provides the data and control files on it. What the application can do with these files depends on the type of panel, although most operations work for all the panels. Besides `ctl` and `data`, directories representing rows and columns have one extra subdirectory for each one of the panels they contain.

As shown in figure 2, the name of a directory determines the type of panel it represents. A name is of the form *type:name* where *type* is any of the known type names.

The data file contains a portable representation of the panel, text for text elements, Plan 9 images for images, etc. Widget data can be updated by writing this file, and can be retrieved by reading it. For example, the *draw* panel provides vector graphics. Its data file contains a series of drawing commands specified as text. The panel interprets the commands and performs them. The clock shown in figure 1 uses a single draw panel.

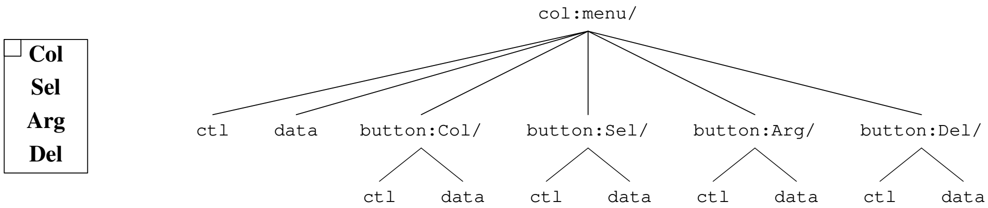

Figure 2. The file tree for the omero menu shown in most user screens.

The `ctl` file contains a textual representation of the panel attributes. This is an example `ctl` file for a text panel:

```
addr tcp!nautilus!17218
notag
show
dirty
font R
mark 28689
sel 28067 28067
size 65 39
```

The first attribute in the example is the network address where omero delivers events for the panel. When this attribute is set, omero dials the given address and starts delivering events. If the connection breaks, omero assumes that the application has exited and removes the panel. The `notag` attribute and its complement, `tag`, determine if the panel is given a tag or not. Panels with a tag can be moved around with the mouse and accept all the mouse commands for tags.

Both files, `ctl` and `data`, are complete descriptions of the panel, (i.e. they are not streams), which means that tools like `tar` or `zip` can be used to copy a hierarchy of panels from one place to another (maybe across different machines), and the resulting GUI would be similar. To permit selective updates of individual attributes, the textual representation for an attribute may be used as a control request by writing it to the `ctl` file.

3.1 Events and event channels

Events are delivered from omero to each application responsible for a panel. All events are delivered as strings and carry the path for the panel involved, the event name, the size of their only argument, and an argument string. Events have a high level of abstraction to easy the portability for the API. The most important events are `look` and `exec`, which usually result from a user's mouse operation. Most applications attend just these two events and ignore others. `Look` is sent when the user asked to look for something. This may be a file name or a piece of text to be found in a panel. `Exec` is sent when the user asked to execute something. The argument for both events is the text involved in the mouse operation, which is sent verbatim to the application. There are several other events used to notify of changes in the data for the panel.

4. Distributed and replicated interfaces.

Creating a distributed interface is easy with omero. Panels can be created at different machines just by creating the corresponding files at different file systems. Because all the panels are the same (i.e. files) and omero accepts the same mouse interface for all of them, the user can move any part of an application's interface to a different place. Furthermore, the user can create with the mouse a row or column and put into it either copies or original controls coming from different applications.

When the user moves a panel using the mouse its corresponding directory moves as well. The `path` event notifies the application of the new position for the files affected. Movements of panels between different machines are handled by replicating the panels at the target and then removing the ones at the origin.

A panel can be replicated by using the mouse to copy and paste it (perhaps at a different machine). The command underlying this operation is actually `tar`, which is used to archive the file hierarchy for the panel and then to extract it at the target directory.

While the files are being extracted, each directory creation causes a new panel to be built. The relative position for the panels extracted is preserved because the omero file system lists files in the order used for the screen layout.

At the point when the control files are extracted, any `addr` attribute set for the original panels will be set for the new ones as well. The update of the `addr` attribute causes omero to establish a connection to the application and to send an `addr` event, notifying of the new replica for the panel and also of its path.

In most UIMS, the application establishes the connection to the UIMS. In our case, omero is the one that dials the application's address to establish the event channel. This simplifies replication, because event connections are established as a side effect of copying a UI file hierarchy. The Plan B `graph` library provides a canned interface for omero applications that maintains the set of replicas for each panel. Applications using the library can ignore any replication of their interfaces.

5. Multimodal and heterogeneous interfaces

Each omero is free to implement the panels `row` and `col` in an appropriate way for the device. For example, on displays with limited screen space, it is sensible to show only one set of controls at a time. The mouse interface can be used to navigate through the hierarchy of rows and columns. It would be also straightforward to port omero for text output devices. In fact, most of the omero interface shown in the screen is just text.

The high level of abstraction in the API permits implementing multimodal interfaces to a limited extent. What matters for the application is that the panels mean the same and the event and file formats remain the same. For example, a server for voice menus can accept the creation of button panels within a hierarchy of columns (or rows). This structure may be handled by the server by reading the button labels to the user and asking him to select an option, perhaps by saying a number. When the user selects a button, the server may deliver an `exec` event to the application as omero does.

Note that the user has a very precise control over the application's interface. For example, part of a given graphical UI could be copied to the file system for a voice interface server. The user can choose which part (i.e. which files to copy), yet the application would be unaware of the replication for the interface. Even though a replica is not even using a graphics device.

6. Donation of screen space

On pervasive environments, it is usually useful to be able to donate screen space to users present in the (physical) space. This can be safely achieved by omero just by changing ownership of a panel to a visiting user. The `chgrp` system command may be used to perform the task, and no further tools are necessary. To reclaim the ownership of the space, the initial owner may simply remove the donated panel from the file system.

7. General purpose tools

A powerful consequence of both using files and mapping the interface elements to them is that general purpose tools can be built to operate on any UI considered. We already mentioned how `tar` is used to copy interfaces. Examples are countless, `rm` can be used to remove them, `ls` can list the panels used, `chgrp` can be used to donate screen space, `iostats` can take statistics on UI usage (as it would do with any other file I/O), etc.

An example is the prototype voice command system, that accepts commands to press arbitrary buttons shown in omero. The command takes the text resulting from speech processing and scans for sentences like *press stop* . At that point, `du` is used to find (files representing) buttons that contain the word of interest, e.g. `stop`, and a write to the button's control file instructs omero to simulate an *exec* on it.

8. Experience and Evaluation

There are several demonstrations and screenshots at `http://lsub.org/ls/demos.html`. Other omero papers at `lsub.org` provide a more detailed evaluation.

We have been using omero for months to perform our daily work. The ability to operate on individual panels, independently of which application they belong to, and to re-group then as desired into another panel, has proven to be invaluable to save screen space on machines with very small screens. For example, screens of PocketPCs can be used to hold just the indispensable controls needed by the user. Note that users (or scripts made by them) implement the policy for deciding what to copy and where to copy. Omero provides the mechanism.

The time needed to copy the UI for the player program shown in figure 1 (on the left, bottom half of the figure) from one machine to another is 365ms. This experiment relies on `tar` to perform the work. The time to remove the UI from a different machine, using `rm`, is 790ms. The time to update one or more attributes that do not involve screen operations was 4ms, using `echo`. To put measurements in context, a delay of 200ms is perceived as instantaneous by the user [3] when using a mouse oriented user interface (400ms for operations involving several screens).

9. Related Work

Research on user interfaces and UIMSs (both for pervasive and traditional environments) has been very intensive and still is. We mention here only the most significative contenders to our work, and leave others behind because the differences with respect to our work fall in one or more of the points stated below.

An important difference between omero and most systems mentioned below is that omero provides a extreme flexibility for users, allowing them to pick up any panel and move it, copy it, or rearrange the set of controls in any way desired. The use of general purpose commands to operate on the UIs is also a big difference between omero and these systems, which rely on more complex formats and require specific purpose tools to operate on the application's UI.

UBI [14] and Migratable UIs [4] support the migration of UIs, like we do. They do not permit using general purpose tools (as we do) and they require the introduction of even more complexity close to the toolkit (e.g., GTK) used by the application. Instead, our approach is to simplify and abstract the service to make migration easy.

Acme [15] is the direct ancestor for omero. Like omero, it provides a programmer's interface accessed through a file system. Also, many of the ideas for the screen layout, mouse processing, and several heuristics are taken from it. Unlike Acme, omero provides a more abstract interface, and takes into account the needs for graphics. Besides, omero permits distributing the user interface.

Systems like Fresco [6], Morphic [11], Gaia [18], and Interactive Workspaces [7], provide middleware components for programming distributed UIs. Unlike omero, they require the application to use the middleware chosen by the platform developers. Omero just requires using files, and therefore we can use general purpose tools on UI elements. Furthermore, is not clean how these systems deal with protection and donation of screen space in a safe way. Our approach, on the other hand, relies on well-known distributed file system technology to authenticate and perform access control for the users. This difference also holds for most component based middlewares for distributed interfaces, (e.g. that in the .NET framework).

There are systems like [13, 9, 5, 2] that use XML or similar declarative descriptions to encode specifications for user interfaces, to permit their adaptation to the peculiarities of the devices used, e.g. screen size. In our case, it is the server that services the screen who is free to adapt the implementation of the provided panels to the needs of the device. For example, Pocket-PCs may show only one outer column/row at a time, or use heuristics to overlay them. Our approach is different in that the high level of abstraction in the interface and its portability makes tools like XML unnecessary. Besides, we use the same approach for *all* other system services [1], they do not.

10. Conclusion

We have described an architecture for organizing system support for user interfaces based on using files to represent UI components. This permits distribution and replication of UI components in a simple way. General purpose tools can be used for UI components as well.

The prototype system, Omero, has been used daily for almost a year. It was used to write this paper.

References

[1] F. J. Ballesteros, E. S. Salvador, K. L. Algara, and G. Guardiola. Plan B: An operating system for ubiquitous computing environments. PerCom, 2006.

[2] T. Browne. *Using declarative descriptions to model user interfaces with MASTERMIND*. Springer-Verlag, 1997.

[3] J. R. Dabrowski and E. V. Munson. Is 100 milliseconds too fast? In *CHI '01: CHI '01 extended abstracts on Human factors in computing systems*, pages 317–318, New York, NY, USA, 2001. ACM Press.

[4] D. Grolaux, P. V. Roy, and J. Vanderdonckt. Migratable user interfaces: Beyond migratory interfaces. *Mobiquitous 2004, The First Annual International Conference on Mobile and Ubiquitous Systems*, pages 422–430, 2004.

[5] T. Hodes and R. Katz. Smart spaces: Entity description and user interface generation for a heterogeneous component-based distributed system, 1998. http://sherry.ifi.unizh/ch/hodes98enabling.html.

[6] P. home page. The fresco project, 2004. http://www.fresco.org.

[7] B. Johanson, A. Fox, and T. Winograd. The interactive workspaces project: Experiences with ubiquitous computing rooms. *IEEE Pervasive Computing Magazine*, April 2002.

[8] T. J. Killian. Processes as files. *Proceedings of the Summer 1984 USENIX Conference*, pages 203–207., 1984.

[9] K. Luyten and K. Coninx. *An XML-based runtime user interface description language for mobile computing devices*, volume 2220. LNCS, Springer, 2001.

[10] K. Luyten, C. Vandervelpen, and K. Coninx. *Migratable User Interface Descriptions in Component-Based Development*, volume 2545. LNCS, Springer, 2002.

[11] J. I. Maloney and R. B. Smith. Directness and liveness in the morphic user interface construction environment. *Proceedings of the 8th annual ACM symposium on User interface and software technology*, pages 21–28, 1995.

[12] B. A. Myers and M. B. Rosson. Survey on user interface programming. *Proceedings of the Conference on Human Factors in Computing Systems*, pages 195–202, 1992.

[13] J. Nichols, B. Myers, K. Litwack, M. Higgins, J. Hughes, and T. Harris. Describing appliance user interfaces abstractly with xml. *Workshop on Developing User Interfaces with XML: Advances on User Interface Description Languages*, 2004.

[14] S. Nylander, M. Bylund, and A. Waern. Ubiquitous service access through adapted user interfaces on multiple devices. *Personal Ubiquitous Computing*, 9(3):123–133, 2005.

[15] R. Pike. Acme: A User Interface for Programmers. *Plan 9 Programmer's manual, 3rd ed.*, vol. 2, 2000.

[16] R. Pike, D. Presotto, S. Dorward, B. Flandrena, K. Thompson, H. Trickey, and P. Winterbottom. Plan 9 from Bell Labs. *Computing Systems*, 8(3):221–254, Summer 1995.

[17] A. Ranganathan, S. Chetan, J. Al-Muhtadi, R. Campbell, and M. Mickunas. Olympus: A high-level programming model for pervasive computing. *Proceedings of 3rd IEEE Intl. Conf. on Pervasive Computing and Communications*, pages 7–16, 2005.

[18] M. Roman. *An Application Framework for Active Spaces*. University of Illinois at Urbana-Champaign, 2003.

[19] C. Young, L. YN, T. Szymanski, J. Reppy, R. Pike, G. Narlikar, S. Mullender, and E. Grosse. Protium, an infrastructure for partitioned applications. In *Proceedings of the Eighth IEEE Workshop on Hot Topics in Operating Systems (HotOS)*, pages 41–46, Schloss Elmau, Germany, 2001.