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 compoments 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., [15, 14, 12, 11, 16, 7, 20]. 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. Doing this requires several different frameworks that have to be programmed separately in a different way. Applications cannot usually create a control panel that works both for the phone and for the audio interface.

It is hard to split and merge the UI and place different parts of it on whatever device is considered appropriate. At most, we can combine a set of UI components by using middleware. However, once programmed, we cannot split a given component into separate ones. For example, it is usually not feasible for a user to move just the "next slide" button to a mobile phone.

Replication of UI components is hard and requires collaboration from the application. Considering that it is now usual to have multiple displays and I/O devices, it is desirable to be able to replicate, say, a volume level gauge, 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 [25]. A related problem is that it is hard to consult and update information about the UI itself; for example, obtaining the label for a button from a different program or updating the label to something else.

While constructing the Plan B OS [3, 2], 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 [10, 23]. 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.

The Plan B's UI service, *Omero*, was built using this approach and has been in use in production for half a year. It has been used only by a few users on Plan B terminals, but its approach can be applied to any other system considered.

In what follows, we describe both omero and our approach. Section 2 describes omero from the user's point of view. Section 3 describes our approach in more detail. Sections 4 discusses the replication and distribution of UIs using omero. Multimodal UIs and heterogeneity are addressed in section 5. We address screen space donation in section 6, and the use of general purpose tools in section 7. Sections 8 and 9 describe the implementation and evaluation for the system. After discussing related work in section 10, we conclude in section 11.

2. Omero

Omero is the Plan B window system and the User Interface service. The current implementation works both on Plan 9 [23] and Plan B [3, 2]. 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 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*. Rows and columns group inner panels and handle their layout. A row arranges for inner panels to be disposed in a row. A column does what can be expected. Atoms include text, images, gauges, and the like.

All panels, including rows and columns, are considered the same by omero, like in Morphic [13]. 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.

In the same way, all text shown by omero is considered the same (although some text may be read-only and cannot be edited), following the design of Acme [21]. This includes buttons, labels, tag-lines, and text frames. As a result, we can type some text and use it as a button, we can copy text from buttons or labels and paste it at some other place, etc.

The interaction with the mouse and the keyboard happens within omero, without the intervention of the application. Therefore, most text editing and mouse actions are handled by the same program, omero, and consequently all the edition process feels the same (no matter the application). The mode of interaction is very similar to that of Acme [21], which can be considered a direct ancestor for omero.

Any panel may have a *tag* (a square near its top-left corner). By default, rows and columns have tags, and atoms do not. This can be changed through the file system interface. The tag permits certain mouse operations in the panel, and provides information about the data shown on it. When a panel has hidden panels within it, its tag is shown as a vertical rectangle instead of a square box. A panel may be in a *dirty* state, when the application using it considers that it has unsaved state. In this case, the tag is shown in a light green color.

A central client program for omero is `ox`, which is responsible for file editing, directory browsing, and command execution. It can be considered the shell underlying omero. Figure 2 shows a omero screen with three panels (below the row at the top) created by ox: An edition for `omero.ms`, a listing for the directory `/usr/nemo`, and the output for the `View` command. Using a separate program instead of implementing this functionality within omero permits executing omero at one machine and ox at another. This is common when

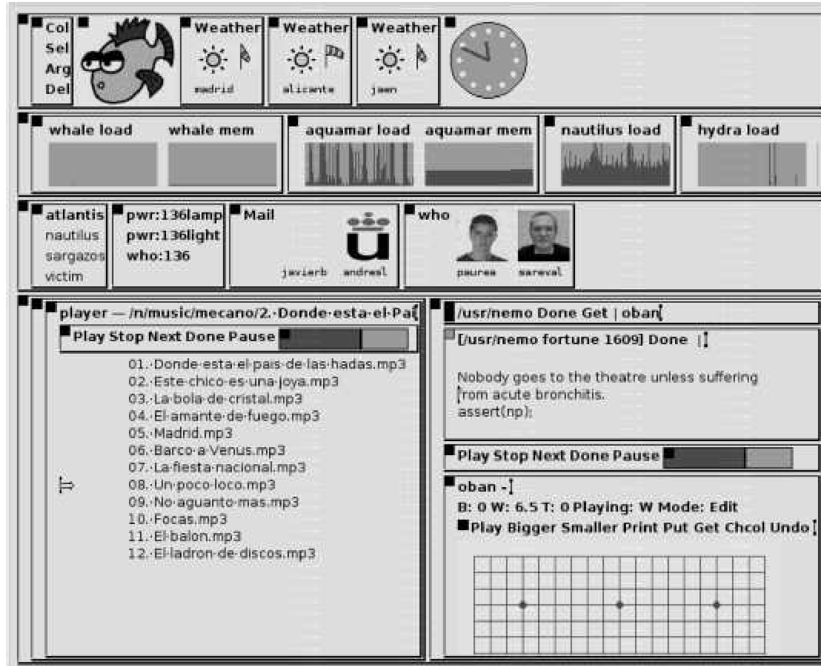

Figure 1. A typical Plan B screen, serviced by the `omero` UI service.

accessing the system from devices like PDAs that may have slow connections: Commands actually run at a machine well connected to the rest of the system and we keep just the interface at the end of the connection. The idea is not new, but taken from Sam [22] and its ancestors.

For each file being edited, ox creates a column that has a tag panel (single line of editable text, in bold face) and a text panel below the tag that shows the file. Ox initializes the tag to contain the name of the file being edited, some commands understood by ox, a vertical bar, and space for the user to type further text.

To help programs started from ox. Ox sets the variable `$file` to the path of the file being edited in the ox's panel where the command execution was requested. This permits the creation of programs that operate on the file being edited, like `View`, which opens a viewer to see `$file` as it would be sent to a printer.

3. Architecture and ideas

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. Rearranging the tree (e.g. due to a user's mouse operation) requires the application to learn of the new paths for its UI files, but it has the benefit of expressing layouts in a simple way by means of the file hierarchy.

The file tree is exported using the 9P file system protocol [1] and can be mounted from anywhere in the network. Several devices supporting UIs can be used together by mounting their respective servers. Applications, like the client process in figure 3, operate the service by mounting one or more omero's file systems (small trees in the figure) on their name space and using the standard file operations.

Much of the design effort went on deciding how to represent widgets as files. Many designs would match what has been said so far. To narrow the design space, we imposed two requirements:

- It should be simple to create and operate an UI from the system shell.

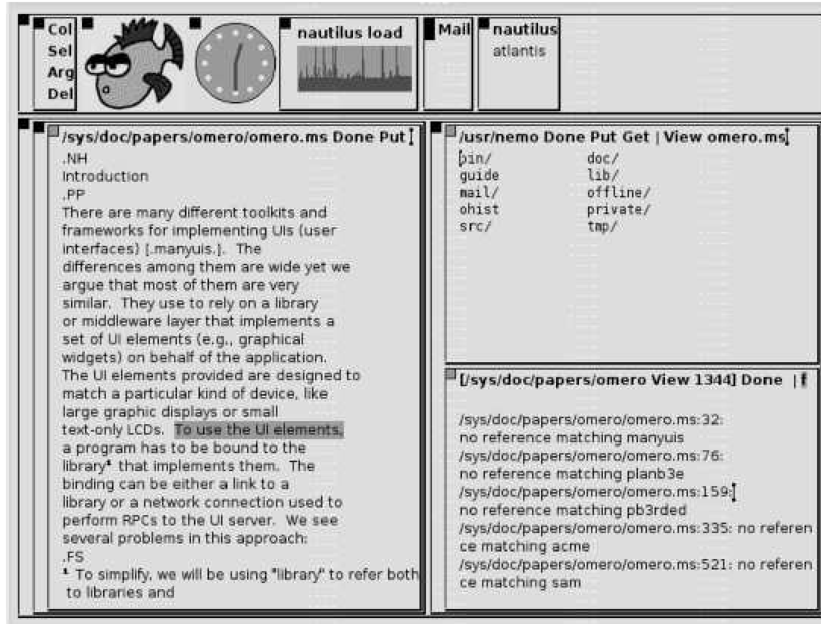

Figure 2. Ox creates panels to edit files, browse directories, and show command output.

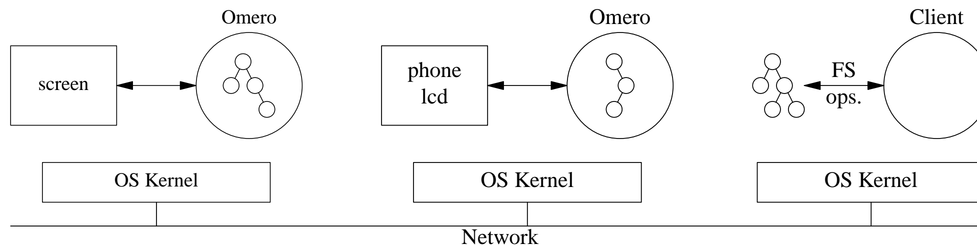

Figure 3. Omero serves widgets as files.

- It should be simple to split, merge, move, and copy an interface to a different device.

The first requirement suggested us that in those cases where the file name could easily express what widget we want, we should rely just on the file name to determine the widget properties. The second requirement suggested that the design should permit a precise replication of a given UI by copying its file tree to a different server. This lead to the file system design guidelines that we show next.

- No streaming data may come out of an UI file. Otherwise, reading that file during a copy operation would block the copy, because a read would wait for further data on the stream. This does not mean that widgets from streaming data are forbidden, only that their corresponding files cannot be the source of data. No events may come out of an UI file, because of the same reason[1]

- Reading an UI file must return all its state. The state must include whatever is necessary to recreate the involved widget, with the same attributes, at a different location.

- Writing an UI file must allow updating all the properties of the widget involved. Otherwise, the interface could not be recreated by copying.

[1] UIs generate events, of course, but events have to be sent through a different channel. Omero uses a network connection independent of the file system to deliver events, as discussed later.

- Directory listing order must be coherent with the creation order of inner files. Or a copied UI might lead a different layout of widgets, because we decided to maintain the correspondence between the file hierarchy and the screen layout.

3.1. Widgets as files

Omero represents each panel by a directory that contains a `ctl` and a `data` file (see figure 4). 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.

The name of a directory determines the type of panel it represents. A name is of the form *type:name* (eg. `text:ox.3442`) where *type* is any of the type names shown in table 1. Usually, *name* is a string randomized by the application to permit any two names to share the same directory (i.e., to share the same container panel).

Panel type	Description
row	Groups inner panels in a row
col	Groups inner panels in a column
text	Editable text frame
label	Read only, single line of text
button	Read only, single line of text
tag	Editable text line
image	Fixed size image in Plan 9 format
gauge	Graphical gauge for a value in [0,100]
slider	Editable gauge
page	Variable size image with panning
draw	Vector graphics

Table 1. Types of panels in Omero.

The data file contains a portable representation of the panel, text for text elements and Plan 9 images for images. The `ctl` file contains a textual representation of the panel attributes. Both files are complete descriptions (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 control file.

Besides the two files mentioned above, directories representing rows and columns have one extra subdirectory for each one of the panels they contain. The order of the files contained in a directory is representative and corresponds to the order used to show their panels in the screen, which is usually the order of their creation. The order in the screen is left to right for rows and top to down for columns. As an example, figure 4 shows a typical menu in omero and its corresponding file tree.

To complete the discussion of the file system interface, we describe now the `text` and `draw` panels, that are similar to other panels not discussed here.

3.2. The text panel

The data file for a text panel appears to contain *all* the text being edited in the panel, not just the portion shown in the screen. In this way, omero can perform most of the editing locally. Operations like undo and redo, inserting and deleting text, cut, paste, and so on, are performed by omero without the intervention of the application. Copying some text into the data file causes omero to reload all the text in the panel. To retrieve the text from omero (e.g., after an edition), the data file is read. To help with programming, the initial text for buttons and labels is set by omero to be just their names (without the type prefix, as seen in figure 4).

The control file for a text panel contains one line for each attribute of the panel. But for `size`, all the other attributes may be changed by a write to the control file using the same format. This is an example `ctl` file:

```
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

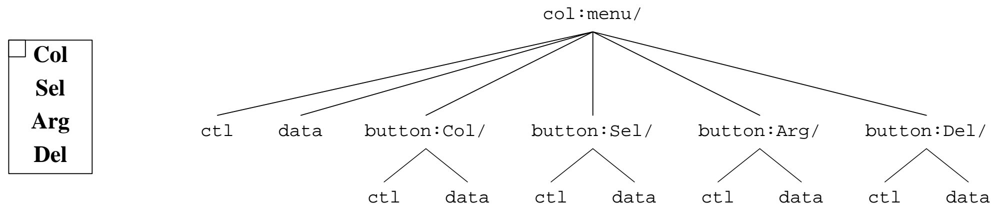

Figure 4. The file tree for the omero menu shown in most user screens.

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.

3.3. The draw panel

The `draw` panel supports drawing of vector graphics. Its control file is similar to the one of the previous section, but its data file is very different. The data file contains a series of drawing commands specified as text. The panel interprets the commands and performs them. For example, copying this text to the file would draw a simple analog clock, similar to the one in figure 1:

```
ellipse 40 40 35 35 grey
line 40 40 20 57 1 blue
line 40 40 52 28 1 blue
```

In this panel we see an example of how omero tries to maintain a high level of abstraction for its interface. These commands for drawing are easily implemented on any platform supporting graphics, and the format used by the application to issue the requests is very portable.

3.4 Events and event channels

Events shown at table 2 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. As we will see, 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.

Event	Argument	Description
look	text	Look for the argument
exec	text	Execute the argument
data	value	Reports new data in the panel
ins	pos text	Text inserted at position
del	pos n	N runes of text deleted at pos.
click	x y buttons	Mouse event
keys	text	The keys in text were pressed
addr	netaddr	Network addr. was set
path	newpath	The panel moved to a new path
exit	-	Panel deleted by the user

Table 2. Events sent by Omero.

There are several events used to notify of changes in the data for the panel. `Data` is sent for simple panels like sliders, and reports the changes in the value (which is also available through the file system). Some panels (e.g. `image` and `draw`) deliver all keyboard and mouse events to the application. The corresponding omero events are `click` and `keys`. Other events are discussed later.

The *event channel* used to deliver events is a network connection established from omero to those applications that request event reception (by setting the `addr` attribute). We do not use the file system interface to deliver events to avoid stream-like files within the UI file system. The event channel between the application and omero is also used to garbage collect the interface for dead applications.

An interesting point is that it is useful to create panels without setting the `addr` attribute. These panels

stay around forever, until the user deletes them with a mouse operation, or a file system command. The omero image viewer uses this technique to load an image into omero and exit. Once the image is loaded, there is no point in keeping the viewer around. This technique is used by several other programs, including our mail reader and a weather information program (seen at the top of figure 1).

3.5. Clipboard and selection

The user clipboard and selection are maintained by a different file server, Omero uses `/dev/snarf` as the clipboard, writes on it the relevant data from cut or copy operations, and reads from it the data for paste operations. This file is usually shared among Plan B terminals for the same user, which means that operations that involve the clipboard, may be performed across different machines.

The file `/dev/sel` is updated by *omero* with the file system path for the last panel where some text was selected by the user. This is a helper for implementing external commands that operate on selected text regions.

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 UIMSs, 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 application can handle the replication following three simple guidelines:

- To read a file from omero, it can be read from any of the replicas.
- To write a file into omero, it must be written at all the replicas. This also applies for creating and removing files.
- When omero reports using an event that data changed in a panel, the other replicas must be updated with the data change reported in the event.

The Plan B `graph` library provides a canned interface for omero that operates in this way. The library handles `addr` and `path` events to maintain the set of replicas for each panel. Applications using the library can ignore any replication of their interfaces. The library also creates a network listener process to accept event connections from omeros, and mounts those omeros whose files are not found in the application's name space. Doing so permits the replication of UI components into omeros not yet known by the application.

To improve latency in the replication of text panels (which may hold large text files), `Ins` and `del` events are sent to report the insertion and deletion of text. The `graph` library handles these events as well. They lead to updates for other replicas of the panel being edited.

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 `ofiles` shell script, which lists the files being edited by the user. This script, shown in figure 5, recursively lists the files in omero to find `tag` panels, which by convention contain the names for the files being edited. For each `tag` panel, its data file is read to retrieve the name of the user's file.

As another example, a prototype voice command system, that we built for Plan B, 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 (like in the script in figure 5) to find 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.

Note how the applications owning the panels affected may run unaware of any of the external commands used on them. No code must be included to provide support for these commands, because all that is needed is to be able to use files.

8. Implementation

Omero is implemented by 6340 lines of C code and two libraries. One of the libraries (taken from Plan 9) provides support for text frames and has 1231 lines of C code. The other one (adapted from one in Plan 9) provides support for the file system and has 2085 lines of C code. This makes a total of 9656 lines in C, to give an idea of what it would take to port omero to a different platform.

The structure of the program is shown in figure 6. The `fs` module includes the data structure for the file system (a file tree), the code to speak 9P to export the file system to network clients, and the code to authenticate them. Close to this module is `panel`, that maintains the data structures for the panels in cooperation with the file tree. Individual panels are implemented by modules `text`, `gauge`, `draw`, and `image`. Some of them implement several ones. For example, `text`

```
# Use du to list the trees in all the omeros (/devs/*ui).
# Put in $tagfiles the paths for the data file of all tag panels.
tagfiles=`{du -a /devs/*ui | grep 'tag:.*/data' |  awk '{print $2}'}

# for each file in $tagfiles...
for (f in $tagfiles){
        # ...print the path at the beginning of the file
        sed  's| .*||' < $f
}
```

Figure 5. Retrieving information from the UI using scripts.

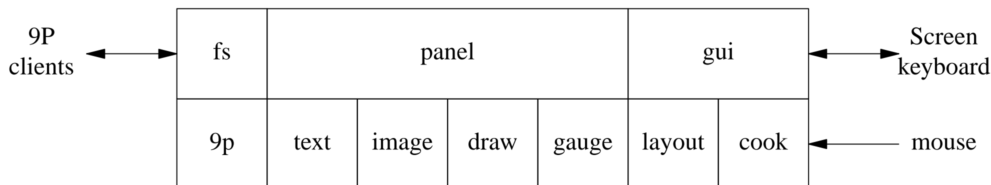

Figure 6. Modules used to implement omero

is responsible for all text panels. Different panels (e.g. buttons and tags) have a different set of flags that determine their behavior (e.g. buttons are read-only, tags can be edited). `Gui` is responsible for handling the screen and accepting user input. The layout processing is hard enough to get right that it is kept separated into the `layout` module.

Mouse input comes from `cook`, which is a module that accepts raw mouse events and provides *cooked* events decorated with time stamps and flags to recognize repeated clicks. The cooking process also supplies `quiet` (virtual) mouse events to notify that the mouse has been quiet for a while.

Panels are implemented by objects that provide these operations: `Init` and `close` are called on panel creation and destruction time. `Ctlio` and `dataio` are called to implement read/write for the panel control and data file. Finally, `mouse` and `keyboard` perform mouse and keyboard processing.

Regarding the client library, it accounts for 1065 lines of C code and provides a handful of routines to create and destroy panels, and to read and write their files. The unconventional part of this library is that it creates a process to listen for omero connections, to receive events from panels and handle replication as said before. The client's data structure for a panel contains a list of replicas and, for each one, the absolute path of the corresponding UI directory. The library handles `addr` and `path` events to keep the list of replicas and their paths updated. Most other events are sent directly to the application through the event channel.

9. Experience and Evaluation

Omero is young and has suffered only minor optimizations, most of them have to do with redrawing in the screen only what is necessary. Overall, we are satisfied both with its behavior and with its performance. There are several demonstrations and screenshots at `http://lsub.org/ls/demos.html`.

Most of the applications we use for daily work have been ported to omero, or rewritten to exploit its benefits. This includes simple tools like clocks and statistics meters, and also more complex programs like mail readers, audio players, image viewers, several games, and other tools.

When used on several machines that share the same room, omero integrates nicely with the Plan B facility to redirect mouse and keyboard devices to different machines. Our users are accustomed to copy omero panels at one machine, then redirect (pressing a button) the mouse and keyboard to another machine, and then paste the panels there.

The ability to operate on individual panels, indepen-

dently of which application they belong to, and to regroup 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.

Regarding quantitative evaluation, it is hard to compare omero with other systems, because of the different approach it follows, and also because the implementation we have runs on a different operating system. Therefore, comparative experiments would be measuring differences between the systems involved, and not between omero and other UI systems. Nevertheless, we include some measures below to give a glance of how it performs.

All measures are the mean of several experiments performed on a Pentium Xeon and a Pentium 6 connected through 100Mbps ethernet. Both machines were running a standard Plan B system, including omero. No machine had swap configured. Experiments involving only one omero were performed across the network, because that is its common usage. All measures correspond to real time measured with `time`, and account for all the relevant screen update operations, because omero performs them before replying to the client programs. To put measurements in context, a delay of 200ms is perceived as instantaneous by the user [5] when using a mouse oriented user interface (400ms for operations involving several screens).

The time needed to copy the user interface 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 uses the standard omero script for copying panels, which relies on `tar` to perform the work. The time needed to read all the UI from the file system was 60ms, which leaves 305ms as the time used to create the interface and update the screen. This is what could be expected, because the creation of an interface involves rather slow screen operations, while accessing the data for the interface does not.

The time to remove the same interface from a different machine, using `rm`, is 790ms. It is higher than the time to create the interface because most panel deletions lead to layout recalculations, while some panel creations can be performed without recomputing the layout (and updating the screen!).

The time to hide one of the `Weather` panels of figure 1 (near the top) is 8ms. The command was `echo hide` with stdout redirected to the panel's control file. The time to show the panel again is also 8ms. Most of the time is spent on the screen updating. If we operate on one of two large columns shown in figure 2, 6ms are needed for `hide` and 8ms for `show`.

The time to update an attribute which does not involve screen operations was 4ms when using `echo` and 7ms when using `cp` to perform the copy of the new attribute values. This time was independent of the number of such attributes updated by the request (the measures involved up to 10 attribute updates per request). This suggests that the performance of omero is reasonable for attribute handling, because other factors (e.g. the program used to copy new data) are more significative in the measures.

The performance of omero seems reasonable to provide user interfaces, as our experience using it for daily work during the past six months confirms. Most of the latency comes from the series of RPCs generated by the underlying system to satisfy the file system calls made by the application. The time to update the screen seems to be the dominant factor, as it could be expected.

10. 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 [17] and Migratable UIs [6] 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 [21] 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.

Omero takes from both Sam [22], the Blit [19] and Protium [29] the idea of separating the program processing from the user interface. The separation of the program into ox and omero results from this. Sam is only a text editor, and does not provide a general window system. The blit was a window system for UNIX, to multiplex a terminal, and was heavily tied to the model of operation in UNIX: A single server machine with terminals connected. Omero on the other hand permits distributed user interfaces and is more flexible in letting the user control the applications interfaces. Protium uses a rather different approach and does not allow general purpose tools to be used on UI components.

Regarding window systems, X [27] and Photon [24] permit applications to create UI components remotely, but their API is rather low-level, unlike in omero. Furthermore, once created, UI components are tied to the particular server used and cannot move. The same happens to window systems like Rio and its ancestor $8\frac{1}{2}$ [1, 20], which despite using files to provide their APIs, do not consider mobility and replication for user interfaces nor distribution of the application's interface (further than done in X).

Toolkits like GTK+ [28], Tcl/Tk [18], simplify the construction of the application's UI, but still lead to the same difficulties mentioned above. Another difference between our approach and these systems is that it is not feasible to implement browser applications just by loading the relevant information into the window system and then exiting.

Systems like Fresco [8], Morphic [13], Gaia [26], and Interactive Workspaces [9], provide middleware components for programming distributed UIs. Unlike omero, they require the application to use the middleware choosen 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 [16, 11, 7, 4] 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 [3], they do not.

11. Conclusions and Future Work

We have described an architecture for organizing system support for user interfaces based on using files to represent the UI components. We have shown how this can permit distribution and replication of UI components in a simple way. The architecture makes it easier to use different devices at the same time and permits the user to split and merge UI components without placing the burden in the application. We did show how general purpose tools can be used for UI components as well. The implementation for a working system that uses this approach (used to write this paper) has been described as well, together with several examples of use.

In the future it would be interesting to explore how to use our approach for applications that use conventional UI toolkits, to provide backward compatibility for them. The heuristics used by omero may also require further experimentation. Porting omero to other systems is also desirable.

References

[1] Plan 9 programmer's manual. *AT&T Bell Laboratories. Murray Hill, NJ.*, 2000.

[2] F. J. Ballesteros, K. L. Algara, G. G. Muzquiz, and E. Soriano. The design and implementation of plan b 3rd edition. a dynamic distributed computing environment. *GSYC-TR-2004-05*, 2004.

[3] F. J. Ballesteros, E. S. Salvador, K. L. Algara, and G. Guardiola. Plan b: An operating system for ubiquitous computing environments. *Submitted for publication. Also at http://lsub.org.*, 2005.

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

[5] 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.

[6] 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.

[7] 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.

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

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

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

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

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

[13] 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.

[14] B. A. Myers. User interface software tools. *ACM Transactions on Computer-Human Interaction*, 2(1):64–103, 1995.

[15] 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.

[16] 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.

[17] 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.

[18] J. K. Ousterhout. *Tcl and the Tk Toolkit*. Addison Wesley, 1994.

[19] R. Pike. The blit: A multiplexed graphics terminal. *Bell Labs Technical Journal*, 63(8/2):1607–1531, 1984.

[20] R. Pike. 8, the plan 9 window system. *Proceedings for the Summer USENIX Conference*, pages 257–265, 1991. Nasville.

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

[22] R. Pike. The text editor sam. *Software, Practice, & Experience*, 17(11):813–845, Nov. 1987.

[23] 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.

[24] Qnx. Qnx neutrino os developer support, 2004. http://www.qnx.com/developers/docs/momentics621_docs/momentics.

[25] 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.

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

[27] R. W. Scheifler and J. Gettys. *X Window System*. Digital Press, 1992.

[28] G. G. team. Gtk+, the gimp toolkit, 2004. http://www.gtk.org.

[29] 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.