O/live: Transparent Distribution, Persistence, and Partial Replication for Ubiquitous User Interfaces

Francisco J. Ballesteros [a] Gorka Guardiola [a] Enrique Soriano [a]

[a] *Laboratorio de Sistemas — Universidad Rey Juan Carlos Madrid, Spain.*

Abstract

This paper is focused on the system side of the multidisciplinary problem of building User Interface Management Systems (UIMS) for distributed and heterogeneous I/O devices. It presents a new architecture that decouples applications from their interfaces by using distributed synthetic file system interfaces (similar to /proc on UNIX) to export user interface elements and a new UIMS, O/live, following this approach. The UIMS has been in use for several years on a daily basis, in our laboratory and personal computers. It supports transparent distribution, replication, and migration of user interface elements among highly heterogeneous devices. Moreover, it is highly programmable without the need for special tools, which (i) facilitates experimentation and iteration for new HCI techniques and (ii) enables the creation of orthogonal services to manipulate programmatically and independently the elements of the distributed UI. This paper describes both the approach and the O/live UIMS and Window System.

Key words: Distributed Systems, Pervasive Computing, UI, UIMS.

Email addresses: `nemo@lsub.org` (Francisco J. Ballesteros), `paurea@lsub.org` (Gorka Guardiola), `esoriano@lsub.org` (Enrique Soriano).

[1] This work has been funded in part by the Regional Government of Madrid (CAM) under project CLOUDS S2009/TIC-1692 and Cloud4BigData S2013/ICE-2894 co-funded by FSE & FEDER. This work has been funded in part by the Spanish MINECO project TIN-2007-67353-C02-02 and TIN2013-47030-P.

1 Introduction

User interfaces are central to pervasive computing applications. HCI issues in this environments are particularly challenging. As a prerequisite to address them, we need an architecture which meets several requirements that we describe now.

As it can be seen, we enumerate only the ones we consider most relevant. Along with each one we include an example scenario, addressed by our approach, which is not handled well by other related work.

- **Integration of distributed I/O devices** to support UIs on behalf of users and applications. Today, users have a myriad of I/O devices. Therefore UIs should be able to combine them. Only users know which devices are convenient as an interface each time an application is needed, and it does not really matter where the application runs.

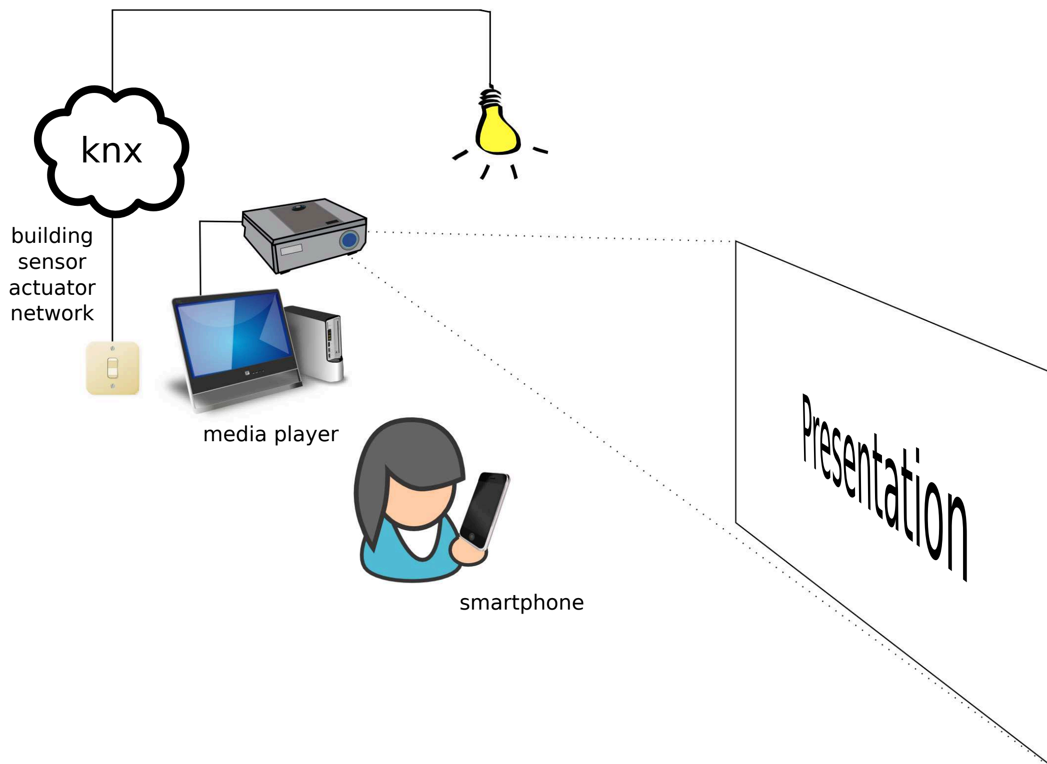

Fig. 1. Scenario, user controlling lights and presentation.

For example, as depicted in figure 1 the user might want to use the touchscreen of a smartphone to dim the lights and then press some buttons to control a video player displayed in a video wall as part of a presentation. In this case, different applications are involved and they are likely to execute on diverse machines. Each one will have its own UI, yet the user might desire to employ the smartphone as the controller for both the light control

appliance and the home video-wall installation.

- **Combination of highly heterogeneous devices.** Besides being distributed, I/O devices are quite different. Compare for example the display used in a desktop computer with the one used in a smartphone. Or compare any of them with a voice controlled device. Applications should be able to exploit these resources without being exposed to low level details of interest only to widget implementors.

  In the previous example, the devices involved may be a smartphone, a system-on-a-chip embedded device controlling a KNX [1] network (a kind of sensor/actuator network infrastructure) acting as the light controller, and a media player device connected to a large display. The applications running in these devices may have their own UIs, probably suited to the I/O devices of their respective machines, which depart from the smartphone I/O capabilities.

  Moreover, it is important to have an architecture which abstracts the details of I/O devices and makes it easy to integrate them into the system. When radically new I/O devices appear like, for example, Siftables [2], it is always unclear how they should interact with regular applications. Abstracting and reifying them as a component of the UI makes it easier to experiment with them.

- **Replication and re-arrangement of widgets at will.** Both the user and the applications should be able to customize and reconfigure UIs, even when it means to split an existing UI (e.g., to move or copy several controls to a different device without moving or copying the rest of the UI). In many cases, this implies replicating widgets (e.g., keeping a copy of the original controls in their original place). Even further, an application can share or borrow widgets from another applications, blurring the limits of what an applications is.

  In the scenario presented, the *smart room controller* shown in the smartphone might be interested only in some parts of the UI of the video player (e.g., elementary play, pause, and stop buttons) and the light controller (e.g., the dimmer bar of the light controller application).

- **Programmatic interfaces for distributed, heterogeneous UIs.** It is clear that many applications will have to be able to inspect and modify their own UIs. Furthermore, for integration and deployment purposes, an application may want to inspect and modify UIs of other applications, perhaps remote. All the operations on widgets, including common interaction (e.g., pressing a button) layout customization, etc. should be provided in a programmatic way.

  In the proposed example, the user or a third party might build a *smart room controller*, simply by combining the relevant widgets of the other applications. Actions like, for example, locating switch-off buttons for lights in the room and pressing them, must be available for external programs.

  Note that this requirement is harder than the *distributed application model* requirement posed by [3], because it requires the UI to be reified and dis-

tributed with independence of the application so it can be controlled by the application, the user or external applications.

- **Neutrality with respect to programming languages and application runtimes.** Today, multiple programming languages and run time platforms are used even within a single device. In many cases, even within the same application. It is desirable for the UI technology to be agnostic in this respect.

  This requirement includes what other authors call *legacy system agnosticism* [3].
- **A distributed security model.** Having distributed programming interfaces requires the means to protect and keep secure the resources exported, in this case the user interfaces. This requires a model for authentication, authorization and access control.

Such requirements are needed not just for achieving targets like plug-and-play computing [4] and pervasive computing in general, but to be able to address the HCI needs on any distributed environment with multiple input methods. Also, some of these requirements (e.g., providing programmatic interfaces) are important to create *low viscosity* [3], which means that the infrastructure facilitates iteration and experimentation of new user interfaces and new HCI methods both at design-time and runtime. This is essential to find new ways to overcome the UI isolation and absence of integration in modern computing environments, where the user has access to multiple devices like smartphones, big shared displays, small embedded devices, etc. and can only use them in an integrated fashion through the use of ad-hoc applications or specially written software.

Devising an UIMS and an architecture satisfying all the issues stated is an example of *deep approach* [5]. Deep approaches seek to directly influence the architecture of the HCI infrastructure itself. They require the engagement of multiple technical disciplines, and as a consequence broaden the scope of the problem, which spans from computer interaction to application and system design. We have centered ourselves mainly, coming from an OS background, in a thorough redesign of the architecture of the way user interfaces are constructed, used by applications and accessed by external programs. To the best of our knowledge, there is no UIMS addressing all the requirements in an integrated and uniform way. This claim is justified in section 3, discussing related work.

We have developed a new architecture to address all this requirements, and built the O/live UIMS using it. This architecture has profound implications for HCI when using it in practice. It allows programmers to create applications that provide the final user with the illusion of a single virtual computer, in which the interfaces are not longer a monolithic block of controls; now, these controls are not confined to a single device or machine, and they do not dictate

a unique mode of interaction for using them.

The basis of our approach is the use of distributed synthetic virtual file systems. Filesystem access is portable, supported in almost all programming languages, and operating systems, has legacy tool and application support and known solutions for access control.

As part of the implementation of the architecture, we have developed a UI which satisfies us as programmers and which we have used also to iterate through different forms of interaction. The reader needs to be warned though, that the user interface, while productive and powerful, lacks the eye candy of commercial operating systems. Adding new viewers to the architecture with either a more traditional look or using completely different and new input methods is not only possible but easy, due to the designed we have followed.

2 The O/live approach

To address the requirements previously described, we have developed a novel architecture and have implemented an actual UIMS and Window system, O/live. It is an evolution of an earlier prototype called Omero [6]. The main idea of our approach is to decouple both the user interaction and the application from the state of the user interface. This scheme facilitates (i) using different displays to access the same interfaces, (ii) combining widgets from different distributed applications, (iii) abstracting widgets to be manipulated through poor network connections, (iv) creating multimodal viewers, and (v) preserving the state of the interfaces between sessions. The ideas underlying the new architecture are simple ones:

- Widgets are represented as an abstract set of synthetic (virtual) networked file hierarchies, describing their state. This idea is depicted in figure 2. Note that by files we mean a file-like API with a small set of operations (open, read, write, etc.) distributed by using a network file system protocol (similar to CIFS, NFS or 9P). Applications operate on these files to update their interfaces.
- In order to manage the state of the interfaces, a file server program provides these "files" and dynamically processes operations over them. The file server coordinates their replication and the concurrent access for both applications and viewers, and provides notification services. UI subtrees may be freely replicated and re-arranged without disturbing the application.
- User interaction, I/O activity, and editing are confined to viewers. Viewers choose which UI trees or subtrees to display by remotely accessing file subtrees. Therefore, viewers are able to operate on remote trees of widgets, and they are free to implement them in an appropriate way for the particular

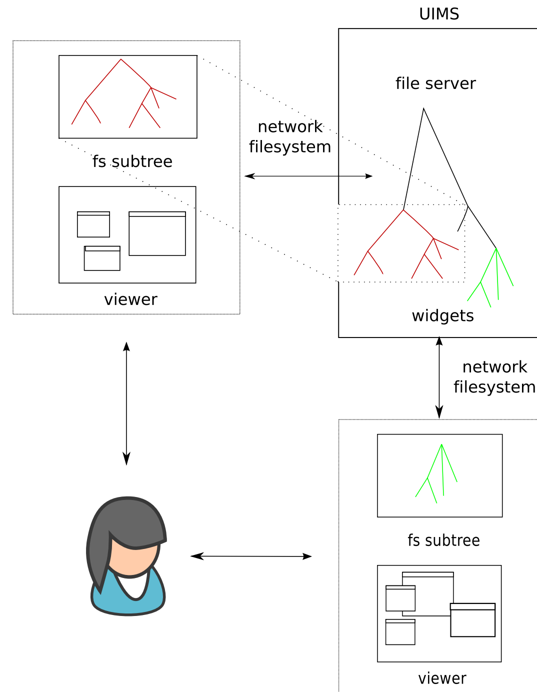

Fig. 2. Widgets are represented by a hierarchy of files. The UIMS is a file server providing files. The viewer mounts a subtree and represents it.

device(s) on which they are deployed.

These ideas can be combined to address the issues enumerated in the introduction, as we briefly show in the rest of this section.

Using distributed files as the interface for building UIs greatly helps portability, because viewers are free to represent data in any way that fits the particular devices they use. For example, a viewer might use voice to represent menus, another might use pixmaps, and another might use text. As a result, it is trivial for the application to support highly heterogeneous I/O devices: It has to do nothing in that respect. The application is only in charge of updating the state of its widgets. Representation and most interaction tasks are transparent for the application.

This approach permits us to easily compose different elements within UI containers, because widgets appear to be files. As such, they can be easily copied, moved, modified, and inspected, and programs can handle them easily. This technique permits us to easily implement a widget editing language to operate on distributed elements, similar to the Sam language [7,8], with the advantage of managing elements from different UIs that may be in different machines.

Applications may operate on network file trees that represent widgets, unaware of their replication. A central program coordinates the replication of these files providing different file trees used by UI viewers. For the application, there is

no replication, distribution, or rearrangement of (parts of) its UI.

Viewers interface between I/O devices and the UIMS. They interact with the UIs by accessing a per-viewer file tree that represents them, and they may remain mostly unaware of replication, but for a few details. Sections 5 and 6 describe the architecture and provide more details about the distribution of the system.

In the scenario described in the introduction, the cellphone and the media player may be both running viewers. The player application running, for example on the user laptop can have parts of its interface replicated on both viewers which would mount the central file server by using a network file system protocol. The player application would be oblivious of the existence of the viewers. The interactions between the viewers and the UIMS and between the UIMS and the application are mediated by the distributed file system interface.

This approach, using distributed file systems as programmatic interfaces to distribute resources has many benefits in general, as has been known for a while in the distributed systems field, see for example [9], but it has extra benefits to represent widgets and user interface interaction. We summarize all them here.

The first and most important is portability. All programming languages provide support for using files[2]. Therefore, they already provide support for UI programming on O/live, because the interface is just a file system. Applications do not depend on any UI library whatsoever. Moreover, users are able to inspect the widgets just by browsing the file tree, so they do not depend on specific tools. Of course, there must be system support to access remote files, but most operating systems provide it (e.g., WebDAV is supported by a wide range of conventional and mobile systems). For example, despite draconian file access restrictions in iOS (i.e., iPhone and iPad), it permits accessing files from the network (e.g., using WebDAV[3]), which means that it would be able to access files from O/live. Therefore, it would be feasible to implement viewers in these devices. When the operating system does not provide native support to access remote files, a library can be used (e.g., a WebDAV client library).

Related to portability is *legacy tool and application support*, any shell scripting environment and file-manipulation tool can be used to deal with them. Simple scripts and file utilities are now able to operate on UIs and widgets. For example, UNIX file tools like *du* and *find*, both of which are commands that

[2] See for example `http://rosettacode.org/wiki/File_IO`.
[3] See for example GoodReader for iOS, which uses WebDAV to import and export documents.

transverse a file hierarchy, may be used to locate buttons. Simple programs like the UNIX program *echo*, which just prints a string, may be used to press them. Most users will not want to write scripts, but for system administrators and programmers this feature allows fast prototyping, automating UI tests, and a simple way to write small but powerful applications. In few words, any application or device capable of manipulating files can be used to control the widgets. Section 7 discusses this point.

Distribution of file systems has well-known solutions. This means that distribution is fairly easy using some form of network file system.

Using a file representation means also that, provided an authorization domain is shared, authentication, authorization and *access control can work like in any other distributed file system.* This means that, because the user interface file system is distributed to the network, a viewer of one user may be able to represent (or not, depending on permissions) parts of the UIs of another user. It is important to remember that this is the programming interface (or maybe the user interface for advanced users). One can easily imagine a simple application for sharing UI subtree, for example by dragging them to an icon of the user. Under the covers, this application would set the permissions.

Our approach does not rely on middleware toolkits and it is not object oriented, yet it is more flexible and easy to program than other UIMS systems that require using well known toolkits. Our experience with the system in the past few years supports this claim. Furthermore, if the file based interface is not suitable for some developers or projects, it is *trivial to create wrapper libraries* to hide file system operations behind an object oriented interface.

The work presented here is in production and we have been using it daily for some years now. You can refer to `http://lsub.org/ls/octopus.html` for source code, documentation, and some demonstration videos.

3 Related work

The literature for UIs and UIMS is very extensive. Due to the space constraints for this paper, we regret to cite only a handful of related systems that we hope are representative enough to permit comparing O/live with those not mentioned.

To the best of our knowledge, this is the only UIMS that allows distributed and highly heterogeneous I/O devices to be freely combined by users to operate on distributed, replicated, widgets in a transparent way for the application. That is the main difference with respect to other related work.

Most related work, including the few systems mentioned here, is either not able to support such a high heterogeneity or not able to do it transparently or does not give users (and programs) the high degree of freedom to, independently, operate on replicated (portions of) distributed widgets. Those that are close to achieving this, require specific middleware or languages and/or specific and complex toolkits [10].

Furthermore, O/live is the only UIMS that we know that provides a highly scriptable and programmable interface based entirely on file-like interfaces, providing, among other things, external names to widgets, protection for them, and enabling the use of general-purpose file tools and scripts to operate on interfaces and interface elements.

Omero [6] is a direct ancestor of the UIMS presented here. In Omero the application relies on a small library that talks to the UIMS. Omero was a departure from other systems in that it provided a file interface and also in that its library coordinated operation of replicated widgets, making the replication transparent for the application. However, it made the mistake of embedding representation of widgets within the UIMS, thus becoming a window system, and not fully decoupling state handling for widgets from their representation. This is important and harms multi-modality and portability. In addition, its design had some drawbacks. The distributed editing required coordination between different Omeros with support from the application's library. In addition, a separate discovery service was required, because all Omeros were peers and there was no place where to look for the list of known devices. Last, Omero was in charge of drawing and accepting input from devices. This means that, although different implementations were feasible, there was no way to separate user interaction from widget management (e.g., as it could be done by following a MVC design). All these issues have been solved in O/live. Either way, we have to say that Omero is an early prototype for O/live, and it must be seen as a previous step in the development of the architecture introduced by this paper.

The Plan 9 window system [11] uses files as the primary interface (like O/live does) but, in most other aspects, it is a traditional and centralized window system. It uses files as an interface for low level operations (e.g., draw a rectangle). Unlike in our approach, it does not include the concept of widget (nor other comparable abstraction) and it has no support for replication or customization of user interfaces.

VNC [12] and similar desktop sharing systems have nothing to do with O/live. They provide viewers for remote virtual framebuffers. In contrast, in O/live, the network dialog involves abstract data, and not pixmaps or mouse and keyboard events. For example, VNC does not work well on WANs due to latency problems; O/live does. Increasing the level of abstraction and using

widgets instead of rectangles helps in this respect. We have been using the system for our daily work and it has encountered a broad range of network bandwidths and latencies (e.g. 1 Mbit/s overseas connections with $\approx$ 150 ms of RTT and home connections with $\approx$ 250 ms of RTT). The system worked well despite the viewer and the central node being connected through all these networks. In addition, VNC does not permit to replicate or rearrange widgets; O/live can do it. Finally, VNC does not enable multimodal viewers, because it works at drawing level.

Wallace et al. [13] propose shared displays supported by introducing "networked application windows", so that multiple input and output devices may be used, perhaps shared. This work derives from VNC. Authors focus on sharing of displays using application-window granularity, which prevents support for highly heterogeneous environments. Unlike us, they do not decouple the application from the implementation of its interface. Also, users can not customize interfaces (because they are still monolithic windows, as in traditional window systems). But in O/live, a user can drag parts of UIs and rearrange them at will, even to a different machine. Though technically what is implemented is pick-and-drop [14], with some minimum support to teleport the pointer it can be trivially converted into hyperdragging [15] or stitching [16]. Actually, we have been using different programs to teleport mouse pointers (some implemented by us, some not) and we normally hyperdrag of parts of the UI.

There has been previous work using a distributed shared data structure model as their foundation. As an example, Recombinant Computing [17] proposes distributing serialized objects acting as proxies or mediators of the interaction. This approach has two different problems. The first is that it is difficult to port to different systems, architectures and frameworks. In their implementation, they use serialized Java objects, which means that, to interact with this infrastructure, a device needs to be able to run the Java methods they contain. Furthermore, data, behavior and programming interface are inextricably tangled in this approach and it is difficult to uncouple them as argued in [3]. The same problems apply to [18].

Another example of distributed shared data structure model is One.world [19]. One.world migrates application data codified as tuples serialized using Java classes. Again, this means that any device which needs to interact with the infrastructure needs to run Java. One.world also depends on broadcast on a local network. Issues like access control, or having a wider scale interaction through the internet is unclear how it would be solved, as their own authors acknowledge.

Closer to our approach is Shared Substance [3]. Shared Substance is completely data oriented, has a tree structure and the distributed interfaces are

organized like a file system, except it is not a file system. Instead, it is an ad hoc composition distributed data model accessed through a custom made middleware which has most of the operations and properties of a file system. Shared substance could easily benefit from our approach, gaining in portability, because its resources and interfaces are almost those of a file system. Another important difference between Shared Substance and the approach of O/live is that it defines a distributed application model, whereas we define a reified distributed UI and UIMS model. O/live clearly separates the UI and UIMS from the application and provides a general portable programming interface that reifies both.

Research on declarative descriptions for user interfaces (e.g., XML based languages like XForms) permits adaptation to peculiarities of devices like display resolution and so on [20–27]. Souchon et al. published a comprehensive review of these languages [28]. A file hierarchy *per se* provides a tree-based scheme and a namespace. Many other systems rely on XML just for achieving these basic requirements. In our approach, names are dynamically validated by the synthetic file system. Therefore, naming convention violation in a file system is analogous to schema violation when using XML. Once hierarchical naming is provided, XML stops being so useful and resorting to simple space-separated text and conventions is enough. In any case, notice that the file-tree approach does not exclude XML on the leaves if this is wanted. Moreover, a file system provides a protection model (file permissions), an introspection mechanism (the file system itself is a dynamic browsable representation of the live state), and a distribution mechanism (network file system protocol). An XML declarative description by itself does not provide any of these features.

There are many systems that use such markup languages to describe the interfaces. For example, SUPPLE [29], TERESA [30] and MARIA [31] exploit this to customize interfaces for heterogeneous devices. Another example is CONSENSUS [32], which uses RIML, a language based on XHTML and XForms. Applications describe their interfaces using RIML. An adaption engine, external to both the application and the user device, is in charge of adapting the RIML description to the specific kind of device. The RIML markup language includes some meta-data to the user interface description, which is exploited by the adaption engine. Unlike these systems, we suggest using synthetic files as an abstract representation for widgets. This greatly simplifies supporting highly heterogeneous interfaces.

XWeb [33] leverages Web technology adding new interfaces for interaction and collaboration. Like O/live, XWeb permits different views of UIs for different platforms and uses widgets as the main abstraction. Unlike O/live, it requires the introduction of new protocols, middleware, and abstractions. Instead, O/live leverages files, already supported in all computing platforms, and successfully addressing important issues like naming, protection, concurrency

control, etc. For example, O/live can rely on file access control lists to control which users can access which widgets. This issue was solved decades ago for files. However, it is not yet clear what the way to do the same for, e.g., XWeb widgets is. As another example, external applications and users can rely on file names to operate on widgets. It is not clear how applications other than XWeb clients can name and use XWeb widgets.

Extensions to X and some of its toolkits for SNAP Computing [4] aim at plug and play computing. They point out that user interfaces cannot be built assuming one user, or one graphics device, or one input device. Interfaces must be able to cope with multiple users, multiple I/O devices. Replication and migration of interfaces is therefore a requirement. The focus of their work is quite different from ours. They do not focus on decoupling applications from the UIMS and viewers and do not operate on abstract and uniform interfaces as O/live does. In contrast, O/live fully decouples the application from the user interfaces. Also, it is unclear if there is a functional prototype already implemented for SNAP Computing.

Systems like Fresco [34] and Morphic [10] provide middleware components for programming distributed UIs. Unlike our work, they require the application to use the middleware chosen by the platform developers. O/live just requires using files, and so, general purpose file tools, the shell, its commands, and any programming language capable of using files (e.g., Python, C++, or Java) can be used to manage widgets.

Systems like UBI and Migratable UIs [35,36] enable migration of UIs. However, they do not provide an external interface to use external tools leveraging migration facilities and they require more complexity at the toolkit level. Instead, we simplify and abstract widgets to make migration and replication easier.

UI façades [37] was built on Metisse [38] and permits users to select and rearrange components from interfaces transparently to the application. The problem with their approach is that independent views cannot be handled independently (for example, a scroll in one would scroll others) and that the abstraction level is not enough to enable use of highly heterogeneous devices.

VPRI's work [39] leads to powerful scripting tools to handle widgets. A huge difference is that O/live enables using the system shell and any previously existing scripting or programming language, as explained before.

HML5 [40] introduced important features for developing Web-based user interfaces. It focuses in the interaction between web browsers, playing a role similar to O/live viewers, and application providers. Our work could benefit from HTML5 features by implementing web viewers for our architecture, which is addressing different issues, as explained above.

In general, Restful [41] approaches are somewhat similar to the file tree interface in that they have a small set of operations with well defined semantics and a name space convention. It still lacks the well-known unified approach for access control and the portability that network file systems provide. The problem is the absence of well known data representation and interfaces for directories and metadata. Even though opening a connection and using GET, POST and other HTTP requests, the tree provided by the file system needs to be created ad-hoc and in many cases this also means parsing service specific HTML or XML. As a consequence, each runtime needs to use its own language-specific binding to address and access these service methods. On the other hand, the interface for accessing files is already present on any programming language for any given operating system, included how to access directories and their metadata, how to add and delete resources, etc. Semantics for concurrent file operations are also well defined and understood on distributed environments. For example, what happens if the root directory of a tree being accessed is deleted.

The list of benefits deriving from the use of files is too large to be detailed here. But it is a crucial difference between O/live and most related works and we can only hope that examples made illustrate the point.

Last but not least, there are many languages for scripting and automation of UI-related tasks, but the one included in O/live permits to operate on all widgets of interest, distributed perhaps overseas [4] , no matter the machine where they are being used, without any code in the application to support the distribution [42]. In contrast, scripting facilities like Applescript operate on a single machine. Section 5.2 provides further details on this issue.

4 UI Elements as files

In O/live, any UI consists of a tree of widgets known as panels. There are two kinds of panels: *groups* (called *rows* and *columns*, after two popular types of groups) and *atoms*. Rows and columns group inner panels and handle their layout. A row/column arranges for inner panels to be disposed in a row/column. All groups are similar and describe how to handle the layout, as a hint to represent the widgets on graphic devices. Atoms include text, tables, buttons, (text) tag-lines, images, gauges, sliders, panning-images, and vector graphics. The particular set of panels is just a detail of our implementation, but not a key point in our approach, which would still work as long as the panels implemented are kept abstract enough.

[4] This feature is possible by doing lazy updates of high-level of abstraction events.

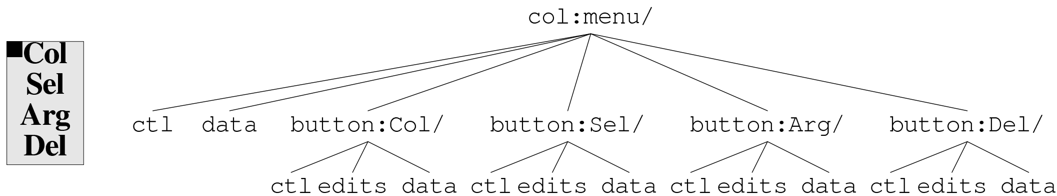

Fig. 3. Widgets are represented by a hierarchy of files. The UIMS is a file server providing files. The figure shows an example menu and its corresponding hierarchy of files.

Each panel is represented by a directory. Panels can be created and deleted programmatically by making and removing such directories. They usually contain two files: `ctl` and `data` (attributes for widgets and their data). Many widgets include another file in their directories, named `edits` (a log of changes). Grouping panels (e.g., rows) contain a subdirectory for each widget they contain. Figure 3 shows a menu and the corresponding file tree. More widgets are shown in figure 4.

To avoid confusion, we must emphasize once more that none of these files are actual files on disk. The same happens with directories. They are synthesized on demand by a file server program, which just happens to be the UIMS. That is, they behave as files but are just an interface to the UIMS. Applications perform operations, mainly read and write operations, over the virtual files and directories to access and control the widgets. These virtual files can be used locally or across the network, just like any other remote file system.

As shown in Figure 3, the name of a directory determines the type of panel it represents. The figure depicts both `col` (columns) and `button` panels.

The `data` file contains an abstract and portable representation of the panel. It contains plain text for text elements, compressed bitmaps for images, textual representation of draw operations for vector graphics, textual representation of a number between 0 and 100 for gauges, and so on. Data for a widget may be updated by writing its data file, and retrieved by reading from it.

The `ctl` (control) file contains a textual representation of the panel attributes. For example, a control file may contain “`sel 5 100`” to state that the text selected in a text widget stands between the 5th and the 100th symbol (unicode character). Another example is “`font T`”, which is a hint for viewers to show text using a constant width (“teletype”) font. 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.

The `edits` file, which is present on many panels, is a textual description of changes made to a panel. This is particularly relevant to text panels and derives (tags, labels, etc.). It is used to maintain a central representation of the history of changes made to a panel. For example, it can be used to undo

operations that were made on a previous session, after reopening it, perhaps using a different terminal if the user has moved from one machine to another.

The exact details of the set of panels and the format for their data and attributes may be found on the user's manual [43]. Here we provide some examples to illustrate the overall scheme.

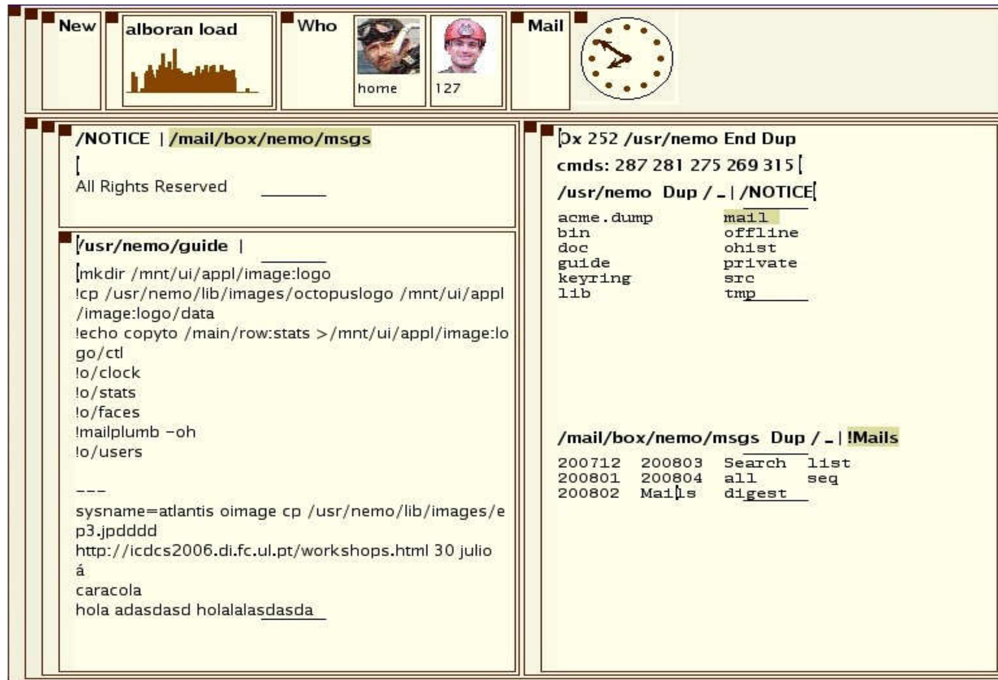

Fig. 4. An example screen running an O/live viewer.

This is the text found in the `data` file for the clock panel shown near the top of Figure 4. Note how we can use the standard *cat* command (or, of course, any other text editor) to read the widget state [5]:

```
term% cat /mnt/ui/appl/draw:clock.8623/data
fillellipse 40 40 35 35 back
ellipse 40 40 35 35
fillellipse 66 40 2 2 mback
        ... other fillellipse lines omitted ...
fillellipse 63 53 2 2 mback
line 40 40 14 34 0 2 1 bord
line 40 40 49 53 0 2 1 bord
fillellipse 40 40 3 3 bord
```

[5] The UNIX command `cat` just writes the content of the file to its standard output. In this example, and all other ones, we assume that the widget hierarchy been mounted at `/mnt/ui`. `term%` is the prompt of the shell in the command-line examples.

The widget contains textual vector graphics commands that may be interpreted by any (graphical) implementation of an O/live viewer. Using this file, the clock application updates the data representation of the panel once per minute.

Different types of widgets may use different data formats for their data files (e.g., plain UTF text, textual vector graphics commands, etc.). Moreover, a given type of widget may be designed either as a container or as an atom. This is a choice that the widget developer must make. Formats shown in this paper are just examples taken from our implementation for O/live.

The widget representation as a file system is up to the developer of the widget. Thus, when adding a new type of widget, the developer can define its language, that is, the data contained in the file(s) that represent the widget. For example, a vector graphics widget could be designed as shown in the paper, or perhaps as a hierarchy of other widgets (analogous to the XML representation of SVG).

These are the contents of the `ctl` file for the same panel:

```
tag
show
appl 0 8623
copyto main/row:stats
```

The first two lines are written (and known) by the application. They instruct O/live to place a tag on the panel (the small square at the top-left corner, which may be operated with the mouse, as discussed later) and to show the panel (it might be hidden).

The third line in the `ctl` file is also written by the application to establish a process id and a panel id on the panel. The former is used by the UIMS to kill the application automatically if the last replica of the panel vanishes (e.g., after the user closes the last instance by using the mouse). The latter may be used to quickly match an event to the panel involved (the name of the panel may be used instead, but it is easier in many cases to rely on a small integer value).

The last line replicates the panel on the main screen, within the row named `row:stats`. Replication will be further explained in following sections.

External programs can be used to update these elements. For example, to make the panel untagged, we can execute in a shell[6]:

```
term% echo notag > /mnt/ui/appl/draw:clock.8623/ctl
```

[6] *Echo* is a UNIX command that just prints its arguments, while > redirects its output to a file.

The same task could be performed by a Python program:

```
outfile = open('/mnt/ui/appl/draw:clock.8623/ctl', 'w')
outfile.write('notag')
outfile.close()
```

Error handling is omitted in the examples for clarity. Note that the file server checks the validity of the data written to its virtual files. In case of error, the system call returns an error and the program can deal with it appropriately.

By leveraging the file abstraction, there are many issues that become easily solved. For example, widgets appear to be files and they have access control lists attached that govern who can or can not operate on them.

Regarding concurrency control, file servers involved in O/live (described in the next section) perform a file operation at a time. This means that client programs may consider file operations as atomic. Also, it means that two different (perhaps conflicting) requests are not concurrent. One of them will be performed before the other, and the last one prevails.

As an example, it is customary for applications to create entire UIs first, and then write a request to a `ctl` file to make the UI visible at a given screen (or device). Because such a write is atomic, the entire UI can be set for viewing in one or more viewers at once, without races.

When updating a data or control file, the implementation takes care of multiplexing requests from different clients (i.e., from different file descriptors). For example, when updating a large image requiring multiple writes, the widget state will not change before closing the file after writing it. Therefore, the update is also atomic as far as clients are concerned. Should two programs update concurrently the same image, one of them will win the race, but the image will be consistent.

5 O/mero and the O/live viewer

O/live has been designed to address the issues discussed in the introduction. Figure 5 depicts its architecture. Unlike most other UI systems, the system is built out of two main programs:

- `O/mero` is the UIMS and provides a single, central, file system interface for all widgets of interest. It is the heart of cooperative and distributed editing and provides support for persistence of widgets across terminal reboots and network partitions. O/mero is designed to be hosted in the cloud. This means that this central point of control can be replicated to provide fault

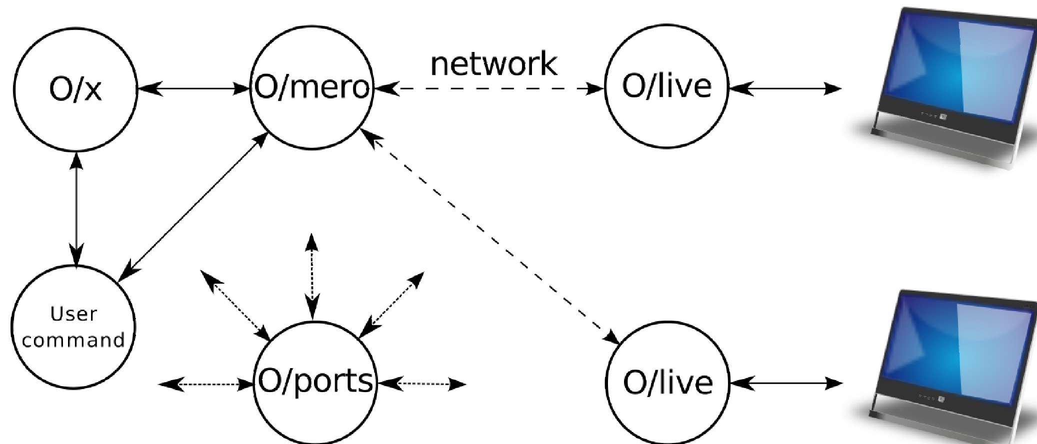

Fig. 5. Main processes involved in O/live: O/mero is the UIMS that keeps the central state for widgets and orchestrates edit operations, and O/x is equivalent to a shell and file browser for the system, including a command language. Components use an event service, O/ports.

tolerance and availability as needed. This program is not to be confused with Omero, an ancestor of O/live.

- `O/live` is a viewer for O/mero (and gives its name to the entire system). It is responsible for all user interaction and it is the only program that knows how to draw and how to process device input (e.g., mouse and keyboard).

This separation between the interface and the machinery for widgets goes back to the Blit [44], a system that provided UIs for UNIX long ago. O/live may be considered a late descendant.

There are two other components in the architecture: O/ports and O/x. O/ports is an event delivery service used by the rest of components. Most of the time, O/live is used in conjunction with O/x. O/x is a file browser, an editor, and a shell to the underlying system. As far as O/live is concerned, it is an application program. It is also in charge of a distributed editing command language, and many applications rely on its services.

As Figure 5 shows, the application interacts only with O/mero using the file interface like in all examples shown. Actual mouse, keyboard, and image processing is performed only within O/live, which may execute close to the I/O devices involved, perhaps overseas.

O/mero's root directory contains a directory named `appl` used by applications to create their UIs and operate them. For example, the clock discussed before created a directory named `draw:clock.8623`, which is a vector graphics panel because its name starts with "`draw:`".

For example, our editor (O/x) does something similar to these shell commands

to create the UI for editing a file with path `"/name/of/the/file"`[7]:

```
term% mkdir /mnt/ui/appl/col:edit1
term% mkdir /mnt/ui/appl/col:edit1/tag:fileinfo
term% echo /name/of/the/file > /mnt/ui/appl/col:edit1/tag:fileinfo/data
term% mkdir /mnt/ui/appl/col:edit1/text:edition
term% cat /name/of/the/file > /mnt/ui/appl/col:edit1/text:edition
```

The same example could be implemented in Python as shown. Note that this Python program does not require any special library to do the job. It just uses the standard module for file system operations.

```
import os
os.mkdir('/mnt/ui/appl/col:edit1')
os.mkdir('/mnt/ui/appl/col:edit1/tag:fileinfo')
outfile = open('/mnt/ui/appl/col:edit1/tag:fileinfo/data', 'w')
outfile.write('/name/of/the/file')
outfile.close()
os.mkdir('/mnt/ui/appl/col:edit1/text:edition')
outfile = open('/mnt/ui/appl/col:edit1/text:edition', 'w')
infile = open('/name/of/the/file', 'r')
for line in infile:
      outfile.write(line)
infile.close()
outfile.close()
```

The application is free to adjust its UI and, once ready, replicate it on any screen. By convention, the application replicates its UI in the same screen in which it has been run. Note that this is similar to asking an UI toolkit to *show* it, and does not require the application to be aware of replication facilities.

The user may change things later in this respect and move and/or relocate any panel (that has a tag). For example, to make our new panels available on the first column of the row used for windows in the main screen, our editor would perform the equivalent of this:

```
term% echo copyto main/row:wins/col:1 > /mnt/ui/appl/col:edit1/ctl
```

The same program in Python would be:

```
import os
outfile = open('/mnt/ui/appl/col:edit1/ctl', 'w')
outfile.write('copyto main/row:wins/col:1')
outfile.close()
```

O/mero processes this request by making the `col:edit1` panel (the file tree

[7] The UNIX command `mkdir` creates a directory in the specified path.

rooted at this directory) also available in the `main/row:wins/col:1` directory (within the O/mero file tree). Note that the state for the UI is kept in a single data structure maintained by O/mero. However, this UI now appears to be present at two different places: `main/row:wins/col:1/col:edit1` and `appl/col:edit1`. This is depicted in figure 6.

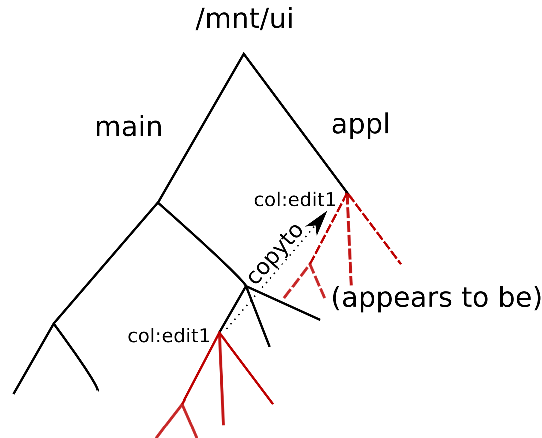

Fig. 6. Copyto replicates a widget subtree, used to show it on a screen.

O/live is started by giving it the path to a screen (or session) that must be shown. For example, executing it with "`main`" as an argument would make it show all panels within `/mnt/ui/main`, including our new example panels.

Should the user want, a new screen may be instantiated by creating a new directory in the top-level directory of O/mero. Previously existing panels may be also replicated on it. For example:

```
term% mkdir /mnt/ui/another
term% echo copyto another/row:wins/col:1 > /mnt/ui/appl/col:edit1/ctl
```

Or, in Python:

```
import os
os.mkdir('/mnt/ui/another')
outfile = open('/mnt/ui/appl/col:edit1/ctl', 'w')
outfile.write('copyto another/row:wins/col:1')
outfile.close()
```

The first command creates a set of panels that conforms the initial layout for a new screen (including a row to contain windows and one or several columns within). The second creates a new replica for our example panels. We have replicated the entire UI element tree that was created by our example. Instead, we might have replicated only a part, by writing to the control file of the directory representing the sub-tree of interest. It is typical to replicate a single button, for example, to make it available on different screens without carrying along the rest of the application's UI.

In the scenario described in the introduction, the UI tree of the video player application would be replicated from the laptop of the user to the media player screen using this. In contrast, only the buttons to play and stop would be replicated to the smartphone screen, making it act as a remote control for the video player. Given the arbitrary decomposition and replication given by this approach, programs can choose what parts of the interface replicate where and this can be changed after the application is programmed by developing new external commands, or directly by advanced users to test new ideas, providing very *low viscosity*.

As another example, this can be used to delete our panels in the C programming language as used on the Plan 9 OS [8] :

```
if(remove("/mnt/ui/appl/col:edit1") < 0)
          fprint(2, "remove failed with error status %r");
```

The important point here is that applications operate on files, unaware of how many replicas there are, and ignorant for the most part of how are panels going to be depicted on the different screens [9] .

On the other hand, O/live (our viewer) is the only one who cares about how to render panels. The same holds for input devices but, before getting into details of input processing, we must discuss event handling.

The decoupling of O/live (viewer) from O/mero enables a N:M relationship between them, unlike in systems such as Omero [6]. For example, one O/mero can serve the widgets to several, multimodal viewers (i.e., to different O/lives). Also, a viewer can combine different O/meros to create a mash-up of more than one application; to do so, it only has to use files from different O/meros.

5.1 Events and clipboards

Most components use event channels, provided by a service named O/ports. O/mero uses this event service to post events for both applications and UI viewers (i.e., O/lives). O/live does not directly post events. Instead, it writes control requests to O/ports and it is O/ports the one that notifies interested parties of any event.

Following the O/live design guidelines, O/ports is actually a file server program. The event service is represented as a directory where files may be created, read, and written. Initially, the directory contains a single file, `post`,

[8] This code just removes the file with the given path and in case of error prints a message on the standard error output

[9] Refer to demo number 2 at [45].

used to post events. Data written to that file is posted to any event channel interested on it. All other files represent event channels.

O/ports is similar to the Plan 9 event dispatcher, Plumber [46], but better suited to dynamic distributed environments (listeners may come and go and event channels are created atomically).

When a program wants to create an event channel and set it up, it suffices for it to create a file on the directory (conventionally mounted at `/mnt/ports`) and then write a regular expression. From this point on, all events with data matching the regular expression will be delivered to the channel. Any further read operation on the file blocks until an event is posted (with data that matches the expression) and then completes conveying the data for the event. Therefore, the reading thread blocks until a matching event is received, but it is trivial to wrap this interface to provide asynchronous event handling if needed.

For example, suppose that some applications need to notify other applications when they save a document. This could be done by posting an event like "`saved:/tmp/c.txt`". To create a listening channel, we could execute this shell command:

```
term% echo '^saved:.*'  >/mnt/ports/savedfiles
```

Any application interested in such an event should read from this file. To further elaborate this example, the following C function can be used by an application to notify other applications that a file has been saved:

```
void
postsaveevent(char *filename)
{
	int fd;
	char data[Maxmsg];

	fd=open("/mnt/ports/post", O_WRONLY);
	snprintf(data, Maxmsg, "saved:%s", filename);
	write(fd, data, strlen(data));
	close(fd);
}
```

or, equivalently in Python:

```
def postsaveevent(filename):
	outfile = open('/mnt/ports/post', 'w')
	outfile.write(filename)
	outfile.close()
```

An example of how event delivery would work in this case is depicted in figure 7

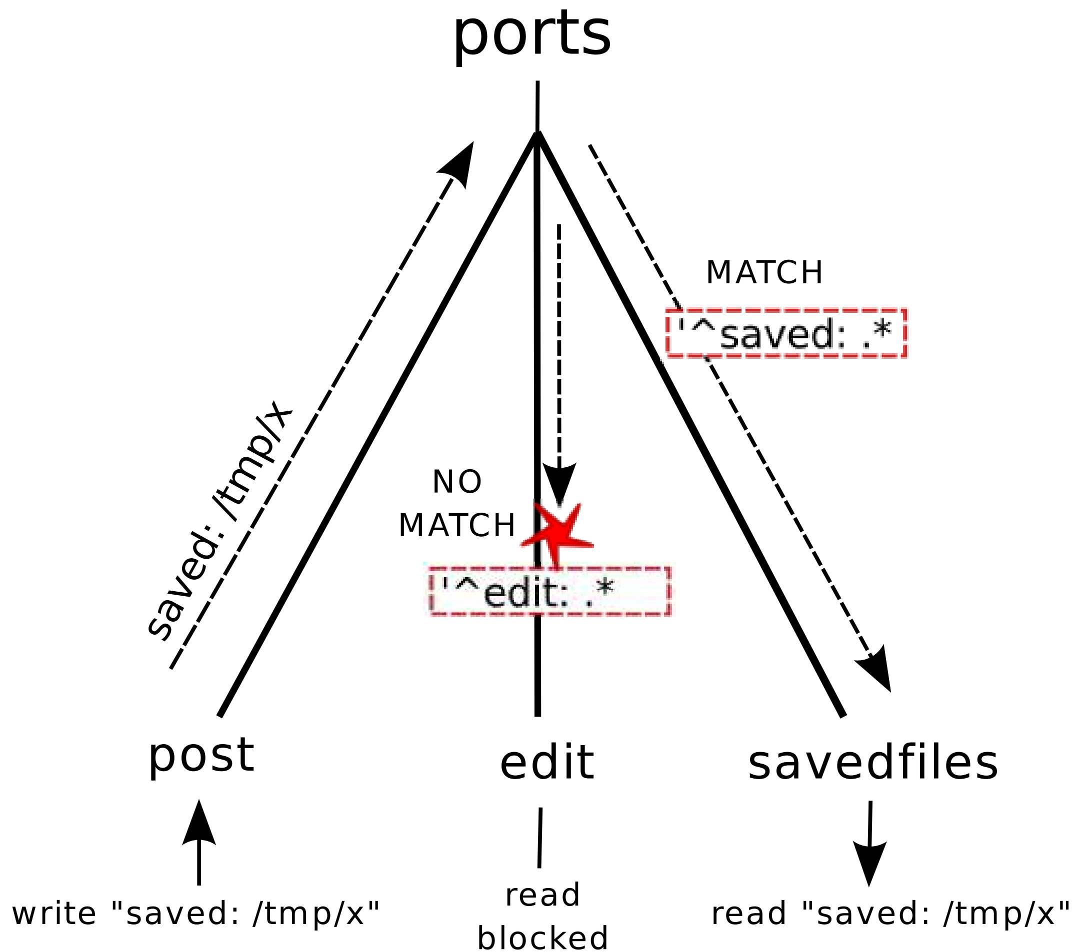

Fig. 7. Event matching example.

Let's now consider (user) focus change events in O/live. When the user establishes the input focus on a panel (by placing the mouse on it), O/live notifies O/mero by writing a "`focus`" request on the panel's `ctl` file, and O/mero posts an event. As an aside, this event indicates that the user is focusing on that UI element, which might have nothing to do with the mouse, perhaps the user relied on voice commands or used a keyboard instead. We can retrieve focus events by creating an event channel, programming it, and then reading from it. For example, we can see focus events involving the panels of the editor (O/x) as shown next.

```
term% echo 'o/mero:.*/col:ox.* focus' >/mnt/ports/focus
term% cat /mnt/ports/focus
o/mero: /appl/col:ox.187/col:ox.ffffd49c/text:file.14 14 focus
o/mero: /appl/col:ox.187/col:ox.ffffb286/text:file.157 157 focus
       ...more output whenever the focus changes...
```

The first command both creates the `focus` port and programs it to retrieve events matching the ones of interest. The second command writes new events to standard output as they arrive.

In the example scenario presented in the introduction section, the presentation application would listen to events representing that the user pressed the button to play the video. Pressing a replica of this button, for example in the phone, would post an event using ports to the application. The application would be listening to events which could be for example, strings matching the regular expression ".*button:Play". In order to do this, it would create a file representing the regular expression as described above and start reading from it. When a button is pressed, the application thread reading from this file would unblock receive the string and start playing the video. As long as all the applications (video presenter, room control), all share the same ports file system, events will be delivered as needed.

Note how, again, external programs may listen for events regarding the UI of any application without disrupting the operation of the application and without application support. Also, no special purpose tools are required; ancient tools such as *cat* and *echo* suffice to handle events programmatically. If this is a security concern, file permissions can be changed for event channel files.

O/live uses a per-user file called `snarf` as the clipboard. Another per-user file, `sel`, contains the path for the panel (in the O/mero file tree) where the selection was made. This is a helper for implementing external commands that operate on selected text regions.

The `snarf` file contains a copy of the user's clipboard. When data is copied to the clipboard, an event is posted. This event is used to update the clipboard on different machines also employed by the user. As a result, we are able to cut some data at a computer, go to a near office, and paste it there on a different O/live.

The full list of events is described in the O/mero manual found in section 4 of [43]. By convention, an event consists of the name of the program posting it, the identifier and path for the panel causing it, a word identifying the event type, and some free form data.

Most events are abstract and not directly related to mouse/keyboard input. There are events notifying that the user started to make changes on a panel, that a panel was deleted (usually using the mouse), etc. The most important events are `look` and `exec`, meaning that the user is looking for something (perhaps wants to search text or to open a new file, depending on the panel) and that the user wants to perform some task (perhaps it was a button push or a command was typed, depending on the panel). There are also low-level events named `click` and `keys` to report raw pointing device and keyboard

activity, for the rare cases where the abstract events are not enough.

5.2 Cooperative and distributed editing

What has been stated suffices to roughly understand how applications interact with O/mero. However, the interaction between O/mero and O/live deserves more explanation. As suggested in figure 5, there may be multiple O/live instances attached to the same O/mero. In fact, that is the case when multiple machines are being used to deploy user interfaces. Usually, there is one O/live per display available (we are referring to the graphical implementation of O/live).

Depending on the user preference, different O/lives may be displaying different "screens" (different file subtrees in `/mnt/ui`, siblings of the `appl` directory), or they might be displaying the same one. This is similar to maintaining different VNC sessions and then using different VNC clients to dial either different sessions or the same one. But the similarity ends there.

Edit operations happen within O/live. For example, an editable text panel implements typical editing facilities within O/live, and notifies O/mero only of text insertions, removals, dirty or clean status change (i.e., unsaved changes), etc. All mouse and keyboard interaction is processed locally, and abstract events are sent from O/live to O/mero, possibly leading to events for the application. The same happens for any kind of interaction with O/live. A consequence of this scheme is that the network link between O/live and O/mero may have high latency (e.g., WAN and 3G networks). The authors have successfully used O/live instances running in America and Australia connected to O/meros running in Europe. By successfully we mean that they were used to perform tasks like writting this paper or debugging the system, without noticing any difference with respect to using it from our laboratory.

In addition, our implementation can be used in conjunction with a network file system protocol designed for high latency links, Op [47], which minimizes the number of round-trip messages involved in file system operations. The Op protocol is out of the scope of this paper and it is described elsewhere [47]. Nevertheless, O/live does not depend on a specific file system protocol. Files can be accessed and reexported using any network file system protocol, such as WebDAV or Styx, but latency can be an issue in that case. For interaction among multiple operating systems or whenever it is convenient, other network file system protocols can be used. Note that O/Live works well either when there is only one user simultaneously using the UI, or when all the users simultaneously manipulating it are connected with low latency to the application. When these requirements are not met, different users may interfere with each

other. In the end, there is no magic: updates take time to cross the network.

All notifications from O/live to O/mero are done through the `data` or `ctl` files of the panels. O/mero processes the request(s) and then notifies involved parties (i.e., applications and other O/lives using the same tree).

O/mero notifies the different instances of O/live of changes to panels by sending notifications whenever a file changes. When O/live receives such notifications, it updates the panels affected. For example, as text is inserted and removed on a text panel, O/live would send `ins` and `del` events by writing control requests to the panel on its `ctl` file. Then, O/mero would update the panel, and notify other O/lives of the insertion/removal or text. They would update their contents accordingly.

The same approach can be followed to implement other types of widgets that control big sets of elements (e.g., lists or tables with thousands of entries or rows) or viewers for devices no able to cache data. Note that the viewers are able to only read the portion of data they are showing. The viewers retrieve the data of the widget from a (synthesized) file. Therefore, they can seek the file and read only the desired segment of data (i.e., the data they are showing). Note that refreshing and layout and size calculations are managed locally to the viewer, and so, in this respect, the limits of our approach are those of a traditional user interface.

When there is an enormous quantity of data to be viewed, and a local cache (in the application) is not feasible, the application needs to know which fragments of data are shown in the viewers (in order to only retrieve these portions from the data source). Although we currently do not support this scenario, it could be supported easily. The viewers would write a command in the control file of the widget to specify the portion of data they are showing. Then, O/mero would be able to send an event to the application specifying the portions of data that have to be loaded in the widget. In addition, an extra attribute would be included in the widget to allow the application to get this information on demand. This way, the application would be able to retrieve the data from the source and load it in the widget. Note that, in this case, the data file of the widget would be similar to a sparse file. Our current implementation does not support this scenario for two reasons. First, we have not needed it. Second, we aim at decoupling the application from UI as much as possible.

For consistency, each panel is assumed to have a version number, which is incremented whenever a change is made to it. Requests to update a panel must supply as an argument the version of the panel before making the change. If the version matches, it means that no other party has modified the panel meanwhile, and the change is made and updates are posted to others. If the version does not match, it was a concurrent edit. In that case, O/mero responds

with an error to the write request (used to write the control file) and O/live (or whoever made the request) learns that it was a concurrent edit.

In our current implementation we discard a concurrent edit when it happens. The approach taken by O/mero and O/live is not to provide failure transparency (i.e., not to resolve conflicts internally when they happen).

If conflicts are rare, it may be fine for the user to type again the tiny text involved in the conflict (in the case of a text panel, it may be a few words). An alternative would be to retry after waiting for the concurrent change that arrived first to O/mero (and made our request fail because its version was not current). Because our implementation does not do that, if there are multiple users editing concurrently and the latency is high, there will be problems. In the case when there is both low latency and an external channel to arbitrate conflicts, the current implementation works well. For example, a group of people in a meeting with private screens (e.g., laptops) and a video wall might talk to each other to resolve conflicts.

O/live is well suited for the two cases it has been designed for: (i) a group of well connected users sharing a common smart space and (ii) a single user using multiple interfaces from local or/and remote locations. In the second case, because there is a single agent using the system, there is no possibility of conflict. Even in that case, there may be several active viewers at the same time at different remote locations.

5.3 Building applications in O/live

It is quite easy to build regular applications using O/live for the interface. Applications only need to create and delete directories, read and write from files and in general interact through the file system with their UI, which they perceive as a single copy.

For example, an application to control the lights of the room 134 which is part of the scenario described in the introduction, would create a button to turn on and off the lights. In order to do this, it has to create the appropriate directory first representing the button with a name like:

```
/mnt/ui/appl/button:light.2342
```

Note that the last number is just a random number to prevent clashes. Then a string like `‘‘appl 34 464’’` has to be written to the file:

```
/mnt/ui/appl/button:light_room134.2342/ctl
```

This string identifies this particular panel and that it belongs to this appli-

cation instance. Writing the string ``copyto screen1/col:1'' would make the button appear on this screen.

Finally a file would be created in /mnt/ports/panelname to receive the events that will be generated for the button. In order to redirect the events to this file, the string ``o/mero: /appl/button:light.2342.*'' would be written to this file. Finally, the application would be in a loop reading from this file. Whenever the read unblocks, the application would turn the lights off and on.

A smart room controller as the one described in the intro would simply create a container. Then it would (again creating a directory) scan all the button directories contained in /mnt/ui looking for the string with the room name and replicate them on the container. This can be done in a simple script or in any programming language without problems as we show in section 7.

The permissions of the files created by the user interface control what the users can do with it. For example, imagine the ctl file for the button described before has read permission for the public, and write permission just for the owner. This means that anyone can check whether the lights are on but just the owner of the file (which may be the owner of the room) might turn them on and off.

Of course, for more complex applications, a small library can wrap the file system operations (which we in fact wrote for C and Limbo) to make dealing with the widgets more convenient. Nevertheless, the main infrastructure to create and deal with files is already present in almost any programming language. This library can look like a normal widget library (for example, GTK+[48]) to the application programmer.

6 Using O/live

This section describes our viewer implementation. It should give an idea of the capabilities of O/live. The specific GUI we have implemented is suitable for advanced programmers, but it is trivial to develop a simple WIMP [10] viewer for the same architecture. Such viewer can be programmed with any toolkit available (e.g., GTK [49]), and there is no need to reimplement the set of widgets.

The mode of interaction and the look and feel of O/live are very similar to that of Acme [7] and Omero [6], which can be considered its direct ancestors. An example screen is shown in figure 4. The interaction is based on the intensive usage of pointing devices. These devices are supposed to have at least two

[10] Window Icon Menu Pointer GUIs such as MS Windows.

(virtual or real) buttons. For instance, on electronic whiteboards, each pen color is considered as a different mouse button. This permits users to use different pens to perform different actions. Likewise, in multitouch devices, different gestures can be interpreted as different buttons.

The interaction with the mouse and the keyboard happens within O/live, without the intervention of the application. Therefore, most text editing and mouse actions are handled by O/live, and consequently any editing feels the same (no matter what the application is).

For example, applications specify whether panels can be individually handled by users (e.g., using the mouse) or not. Viewers (and applications) specify whether panel data is considered dirty (edited, not yet written to wherever the application wants to save the changes) or not. Applications remain unaware of what this means regarding mouse and keyboard handling and unaware of how to represent dirty or clean data. Different versions of O/live may be implemented to provide the same look and feel for the user, independently of applications.

In our implementation of O/live, 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 yellow color (and so are shown the tags of all panels containing the one in a dirty state).

7 UI Programmability

The file system like interface implemented by O/mero together with the conventions used by O/x and other programs to name the files that implement their UIs, makes it easy to program operations on the UI themselves.

The following example script tells the user if she is editing any file or browsing any directory under `/lib`. First, it finds all the files representing the tags from the UI. By convention, the tags contain the names for the files being edited. Note that the tag is similar to the *title bar* in other window systems. Then, it matches the path with the beginning of the file, using a simple regular expression:

```
# Use du to list the trees in o/mero (/mnt/*ui).
# Put in $tagfiles the paths for the data file of all tag panels.
```

```
tagfiles=`{du -a /mnt/*ui | grep 'tag:.*/data' |  awk '{print $2}'`}
if(grep -s '^/lib/' $tagfiles){
        echo 'editing files under /lib'
}
```

The same example in Python:

```
import os, re

# List all the files under path
def du(path):
      fnames = []
      for(dirpath, dirnames, filenames) in os.walk(path):
            for fname in filenames:
                  compfname = os.path.join(dirpath, fname)
                  fnames.append(compfname)
      return fnames

path='/mnt/ui'
for fname in du(path):
      if(re.search('tag:.*/data', fname)):
            tagfname = fname
            for line in open(tagfname):
                  if re.search('^/lib/', line):
                        print 'editing files under /lib'
                        break
```

Examples are countless: *tar* [11] and *echo* can be used to copy interfaces, *rm* can be used to remove them, *ls* can list the panels used, *chown* [12] can be used to donate screen space, *iostats* (a file system I/O statistics tool) can report statistics on UI usage (as it would do with any other file tree), etc.

7.1 A distributed Sam command language

Another new idea in O/live is the use of a command language for programmers [13], adapted from the Sam [8] editor, to operate on distributed user interfaces. The Sam language plays a similar role than the editing languages used in other UNIX editors like `vi` or `ed`. Because all widgets are known to O/mero, independently of their representation, distribution and replication, it is feasible to employ a single command language to handle all of them in a

[11] *tar* is a command which serializes/deserializes file tree trees in UNIX. WinZip could be used to the same effect in Windows.
[12] *chown* changes the owner of a file in UNIX.
[13] Refer to demo number 4 at [45].

similar way Sam handles local resources.

The particular syntax and semantics of the language are not relevant (and are not discussed here). What matters is that the architecture enables its implementation and makes it as simple as it would be for a single, centralized, text editor. It is not a coincidence that the O/live language is indeed a port of the Sam command language (a text editor on Plan 9), but operating now on a distributed set of machines and devices.

For example, this command removes all panels showing C source files from all the (distributed) screens employed by the user; some of the screens involved might be overseas!

```
X/\.[ch]$/D
```

It may seem cryptic, but it means: for all panels ("X") whose tag lines end in ".c" or ".h", delete ("D") them.

This command is also an example of how the different elements in the architecture work together. The user types the command at the terminal using O/live, and such editing happens locally. When return is pressed (or the mouse is used) to execute it, O/live posts an event using O/ports to notify others that the user wants to execute "X/\.[ch]$/D". Neither O/live nor O/ports nor many others know the meaning of the command. For them it is just a text executed by the user. O/x is listening for execute events and honors them. In this case, O/x relies on the file hierarchy (provided by O/mero) to locate files that correspond to widgets and do its job: locating text editing widgets for files with names ending in .c or .h and, for each one, writing requests to delete them.

The language can be used to automate editing (like in Sam), but it includes new loop constructs that operate on panels distributed among the machines used for I/O. It also includes new commands to reclaim panels (copy them) to the screen where the command is issued.

For example, "P/main.*:cmds/e" can be used to copy all panels named "cmds" (usually command buttons) found on the, so called, "main" screen of the user, into the screen where the command is typed. This may be used to establish a replica for all such buttons into a single place immediately. In the example described in the introduction, instead of a special *smart room controller*

In general, it is feasible by typing a similar command to select a subset of the panels by a regular expression, and then to operate on them to perform automated editing, to change their attributes, or to close or move or replicate them.

To the best of our knowledge, the power provided by this command language within O/live has no match in other UIMS and/or window systems found on the literature. At least, when considering that it operates on all panels, no matter where they are.

The most usual task addressed by this feature is to save, close, or collect widgets from applications that were left running at different sessions before continuing work at a different place.

7.2 Multimodal interfaces

To develop a new kind of viewer, neither the application nor O/mero have to be modified. The viewer only has to access the abstract widgets and adapt them to the new interaction. This is similar to what XWeb [33] does.

A prototype voice command system, that we have built for Plan B [50], is a good example of how the programmability of the environment makes things easier. The voice command system accepts commands to press arbitrary buttons shown in the devices surrounding the user. This "system" is actually a shell script, written with a few shell command lines.

The script takes the text resulting from speech processing and scans for sentences like "press stop". At that point, *du*, is used like in the script shown above to find buttons that contain the word of interest (e.g., "stop"), and a write to the button control file instructs O/mero to simulate an *exec* on it.

Note how the applications owning the panels affected may run unaware of any of the external programs used on them. No code has to be included in the applications to provide support for these commands, because all that is needed is already provided by the environment.

In a similar way, it is simple to develop viewers (similar to O/live) for text editing that accept dictation and synthesize voice for hands-free interaction (the viewer may translate voice to text and update the data file for the panel involved). There are many other examples that we omit for brevity.

This prototype system, along with using O/lives with radically different screen sizes (from phone-sized displays to wall mounted ones), permitted us to test if the approach could work well with highly heterogeneous I/O devices or not. It did work well. However, we have to say that only a full viewer has been implemented for graphics devices, the one being used to write this.

8 Experience

Overall, we are satisfied with the environment. Most of the applications we use for daily work have been ported to O/live, 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 simple games, and some other tools.

When used on several machines close by, O/live and O/mero integrate nicely with facilities to redirect mouse and keyboard devices to different machines. Our users are accustomed to copy O/live panels at one machine, then redirect (pressing a button) the mouse and keyboard to another machine, and then paste the panels there.

When the machines are not nearby, the facilities of O/live remain the same. In fact, we are accustomed to employ machines at different locations that feel exactly like part of the same environment, and it seems to us that our "sessions" are always available no matter the devices we use.

The ability to operate on individual panels, independently of which application they belong to, and to re-group them as desired into another panel, has proven to be invaluable to save screen space on machines with very small screens. For example, screens of hand-held devices can be used to hold just the indispensable controls needed by the user.

The performance of O/live and O/mero seems reasonable to provide user interfaces, as our experience using it for daily work during the past few years confirms. By reasonable performance we mean that the system reacts to the user in what is perceived to be instantaneous time and she can forget about the underlying mechanisms. Most of the latency in the interaction comes from the series of RPCs generated by the underlying system to satisfy the file system calls made by the application. Once this latency is minimized by keeping the number of roundtrips low and hidden by containing the interaction in the viewer, the time to update the screen seems to be the dominant factor, as it could be expected. We had to take care of the number of refreshes to make the system responsive enough to be useful.

The experience using a concrete implementation of the architecture, together with the experience iterating with different versions of the viewers porting various applications written in different languages (C, Limbo and shell script mostly) to it and using it, is the best way we know to validate a radically new architectural approach like this. Of course, we could add more viewers of different kinds and port them to different systems, but as it is, we have proved that the approach is sound and fundamentally works and we have built an environment we are comfortable working in.

The proposed architectural approach is, of course, not without limitations. It is not suitable for some kind of applications, such as complex video games. The main problem is that these applications are highly interactive and need intensive, low level graphic control. Nevertheless, this approach works well for most conventional applications. Some applications, like web browsers, appear to be in the same category as video games, but this is not the case. Note that in many toolkits there is a *browser widget* that embeds a browser panel. It would be easy to abstract this widget for our architecture. The state to be kept in O/mero for such a widget would be modest (urls, cookies, passwords, and so on). In this case, the viewer takes care of all the browser interaction, and HTML fetching and rendering. In this implementation, O/mero would not keep any HTML code or graphical state. Something similar would happen with video widgets. The state kept in O/mero would be also minimal (video source, elapsed time, audio track selection, playing status, and so on).

9 Conclusions

To conclude, the key contributions of this work are: (i) the idea of using synthetic files to decouple user interaction from the interface state; (ii) the idea of using synthetic files to decouple the application from its interface state; (iii) a deep approach revisiting how an architecture for UIMSs is built based on the previous two ideas. This architecture is application and language agnostic and supports: integration of distributed I/O devices, combination of highly heterogeneous devices, replication and rearrangement of widgets, and programmatic interfaces for distributed UIs, properties all which; (iv) the description of a working implementation of such architecture and how it enables low viscosity; (v) the report of lessons learned during several years of experience as users of this system.

As future work we are exploring how to use this technology for legacy applications that use conventional toolkits on UNIX and Windows systems to make them believe that they continue to use their UI toolkits, yet make them rely on O/live and O/mero instead. This would provide legacy applications with the same flexibility introduced by our approach. Our current strategy is to intercept calls to their dynamic libraries, to avoid the need for recompiling them. But it is too early to know if this future work will succeed or not.

References

[1] H. Merz, T. Hansemann, C. Houbner, Building automation: communication systems with EIB/KNX LON and BACnet, Springer Verlag, 2009.

[2] D. Merrill, J. Kalanithi, P. Maes, Siftables: towards sensor network user interfaces, in: Proceedings of the 1st international conference on Tangible and embedded interaction, ACM, 2007, pp. 75–78.

[3] T. Gjerlufsen, C. Klokmose, J. Eagan, C. Pillias, M. Beaudouin-Lafon, Shared substance: developing flexible multi-surface applications, in: Proceedings of the 2011 annual conference on Human factors in computing systems, ACM, 2011, pp. 3383–3392.

[4] J. Gettys, Snap computing and the x window system, Linux Symposium (2005) 113.

[5] W. Edwards, M. Newman, E. Poole, The infrastructure problem in hci, in: Proceedings of the 28th international conference on Human factors in computing systems, ACM, 2010, pp. 423–432.

[6] F. J. Ballesteros, G. Guardiola, K. L. Algara, E. Soriano, Omero: Ubiquitous user interfaces in the plan b operating system, IEEE PerCom.

[7] R. Pike, Acme: A user interface for programmers, Proceedings for the Winter USENIX Conference (1994) 223–234.

[8] R. Pike, The text editor sam, Software, Practice, and Experience 17 (11) (1987) 813–845.

[9] R. Pike, D. Ritchie, The styx architecture for distributed systems, Bell Labs Technical Journal 4 (2) (1999) 146–152.

[10] J. I. Maloney, 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 (1995) 21–28.

[11] R. Pike, 8 1/2, the plan 9 window system, Proceedings for the Summer USENIX Conference (1991) 257–265.

[12] T. Richardson, Q. Stafford-Fraser, K. R. Wood, A. Hoppe, Virtual network computing, IEEE Internet Computing 2 (1) (1998) 33–38.

[13] G. Wallace, K. Li, Virtually shared displays and user input devices, in: 2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference, USENIX Association, 2007, pp. 1–6.

[14] J. Rekimoto, Pick-and-drop: a direct manipulation technique for multiple computer environments, in: Proceedings of the 10th annual ACM symposium on User interface software and technology, ACM, 1997, pp. 31–39.

[15] J. Rekimoto, M. Saitoh, Augmented surfaces: a spatially continuous work space for hybrid computing environments, in: Proceedings of the SIGCHI conference on Human factors in computing systems: the CHI is the limit, ACM, 1999, pp. 378–385.

[16] K. Hinckley, G. Ramos, F. Guimbretiere, P. Baudisch, M. Smith, Stitching: pen gestures that span multiple displays, in: Proceedings of the working conference on Advanced visual interfaces, ACM, 2004, pp. 23–31.

[17] M. Newman, S. Izadi, W. Edwards, J. Sedivy, T. Smith, User interfaces when and where they are needed: an infrastructure for recombinant computing, in: Proceedings of the 15th annual ACM symposium on User interface software and technology, ACM, 2002, pp. 171–180.

[18] P. Homburg, L. Van Doorn, M. Van Steen, A. Tanenbaum, W. De Jonge, An object model for flexible distributed systems, Vrije Universiteit.

[19] R. Grimm, One.world: Experiences with a pervasive computing architecture, IEEE Pervasive Computing (2004) 22–30.

[20] K. Luyten, K. Coninx, An XML-based runtime user interface description language for mobile computing devices, Interactive Systems: Design, Specification, and Verification (2001) 1–15.

[21] J. Nichols, B. Myers, K. Litwack, J. H. a. H. M. Higgins, Describing appliance user interfaces abstractly with xml, Workshop on Developing User Interfaces with XML: Advances on User Interface Description Languages.

[22] T. Hodes, R. Katz, Enabling Smart Spaces: Entity Description and User Interface Generation for a Heterogeneous Component-Based Distributed System, 1998.

[23] T. Browne, Using declarative descriptions to model user interfaces with mastermind.

[24] G. Zimmermann, G. Vanderheiden, A. Gilman, Universal remote console prototyping of an emerging xml based alternate user interface access standard, in: Eleventh International World Wide Web Conference, 2002, pp. 7–11.

[25] R. Merrick, Auiml: An xml vocabulary for describing user interfaces.[device independent user interfaces in xml] (2001).

[26] Q. Limbourg, J. Vanderdonckt, B. Michotte, L. Bouillon, M. Florins, D. Trevisan, Usixml: A user interface description language for context-sensitive user interfaces, in: Proceedings of the ACM AVI'2004 Workshop" Developing User Interfaces with XML: Advances on User Interface Description Languages"(Gallipoli, May 25, 2004), Luyten, K., M. Abrams, Limbourg, Q., Vanderdonckt, J.(Eds.), Gallipoli, Citeseer, 2004, pp. 55–62.

[27] J. Plomp, R. Schaefer, W. Mueller, H. Yli-Nikkola, Comparing transcoding tools for use with a generic user interface format, in: Proceedings of extreme markup languages, Vol. 10, Citeseer, 2002.

[28] N. Souchon, J. Vanderdonckt, A review of xml-compliant user interface description languages, Interactive Systems. Design, Specification, and Verification (2003) 391–401.

[29] K. Gajos, D. S. Weld, Supple: automatically generating user interfaces, in: Proceedings of the 9th international conference on Intelligent user interfaces, IUI '04, ACM, New York, NY, USA, 2004, pp. 93–100.

[30] R. Bandelloni, F. Paternò, Flexible interface migration, in: Proceedings of the 9th international conference on Intelligent user interfaces, ACM, 2004, p. 155.

[31] M. Manca, F. Paternò, Extending maria to support distributed user interfaces, in: J. A. Gallud, R. Tesoriero, V. M. Penichet (Eds.), Distributed User Interfaces, Human-Computer Interaction Series, Springer London, 2011, pp. 33–40.

[32] The consensus project.
URL `http://www.consensus-online.org/index.html`

[33] D. R. Olsen, Jr., S. Jefferies, T. Nielsen, W. Moyes, P. Fredrickson, Cross-modal interaction using xweb, in: Proceedings of the 13th annual ACM symposium on User interface software and technology, UIST '00, ACM, New York, NY, USA, 2000, pp. 191–200.

[34] P. home page., The fresco project (2004).

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

[36] K. Luyten, C. Vandervelpen, K. Coninx, Migratable user interface descriptions in component-based development, Interactive Systems: Design, Specification, and Verification (2002) 44–58.

[37] W. Stuerzlinger, O. Chapuis, D. Phillips, N. Roussel, User interface facades: towards fully adaptable user interfaces, Proceedings of the 19th annual ACM symposium on User interface software and technology.

[38] O. Chapuis, N. Roussel, Metisse is not a 3d desktop!, in: Proceedings of the 18th annual ACM symposium on User interface software and technology, ACM, 2005, pp. 13–22.

[39] A. Warth, T. Yamamiya, Y. Oshima, S. Wallace, Toward a more scalable end-user scripting language, VPRI Technical Report TR-2008-001.

[40] HTML 5: A vocabulary and associated APIs for HTML and XHTML, W3C Working Draft (August 2009).

[41] R. T. Fielding, Architectural styles and the design of network-based software architectures, Doctoral dissertation, University of California, Irvine, 2000.

[42] F. J. Ballesteros, P. de las Heras, E. Soriano, S. Lalis, The octopus: Towards building distributed smart spaces by centralizing everything., 2nd International Symposium on Ubiquitous Computing and Ambient Intelligence (UCAMI).

[43] Octopus 2nd. edition user's manual, Laboratorio de Sistemas. URJC. Also at http://lsub.org/sys/oman.

[44] R. Pike, The blit: A multiplexed graphics terminal, Bell Labs Technical Journal 63 (8/2) (1984) 1607–1631.

[45] Lsub demonstrations page, http://lsub.org/ls/demos.html.

[46] R. Pike, Plumbing and other utilities, Proceedings of the USENIX Annual Technical Conference (2000) 159–170.

[47] F. J. Ballesteros, E. Soriano, S. Lalis, G. Guardiola, Improving the performance of styx based services over high latency links, RoSAC-2011-2 Technical Report.

[48] A. Krause, Foundations of GTK+ development, Springer, 2007.

[49] S. Logan, Gtk+ programming in C, Prentice Hall PTR Upper Saddle River, NJ, USA, 2001.

[50] F. J. Ballesteros, E. Soriano, K. L. Algara, G. Guardiola, Plan b: Using files instead of middleware abstractions for pervasive computing environments, IEEE Pervasive Computing, Volume 6, Issue 3, p. 58-65.